View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Aaltodoc Publication Archive

Aalto University
School of Science
Degree Programme in Computer Science and Engineering

Pekka Pöyry

Coverage based safe regression test selection method for Python programs

Master's Thesis
Espoo, September 14, 2017

Supervisor: Assoc. Professor Keijo Heljanko
Advisor: Assoc. Professor Keijo Heljanko

Aalto University
School of Science
Degree Programme in Computer Science and Engineering

ABSTRACT OF
MASTER'S THESIS

Author:	Pekka Pöyry		
Title:	Coverage based safe regression test selection method for Python programs		
Date:	September 14, 2017	**Pages:**	50
Major:	Software Engineering		
Supervisor:	Assoc. Professor Keijo Heljanko		
Advisor:	Assoc. Professor Keijo Heljanko		

Regression testing is a type of testing that aims to verify that the existing test suite will not find any defects in a modified program. Regression tests are usually run after each program modification and may take lots of processing time to complete. Regression test selection is a process where only a relevant subset of tests are selected from the test suite for execution with the goal of reducing the time the regression test execution takes. Safe regression test selection methods are one that can prove that none of the deselected test cases would have found any defects, so that running them is not necessary.

Researchers have proposed multiple different methods for safe and unsafe regression test selection. Many of them require control flow graph or similar information that can be extracted during code compilation step. Therefore most of these methods are unsuitable for dynamically typed programming languages where that information can not be extracted.

This thesis presents a test coverage based regression test selection method that can be used with interpreted programming languages. The presented method does not require any changes to tested program's source code. The presented method's test selection precision was tested with existing medium sized proprietary web application, and the results are somewhat mixed. The overhead imposed by the coverage based test selection method increased the test suite's execution time significantly. The test selection method managed to select a small subset of the test suite roughly half of the time. In the other half of the time the test selection had to re-run all tests.

Keywords:	testing, regression test selection, test coverage
Language:	English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tietotekniikan koulutusohjelma

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä:	Pekka Pöyry		
Työn nimi: Suorituksen kattavuustietojen käyttämiseen perustuva testien valintatapa Python-ohjelmille			
Päiväys:	14. syyskuuta 2017	**Sivumäärä:**	50
Pääaine:	Ohjelmistotekniikka		
Valvoja:	Professori Keijo Heljanko		
Ohjaaja:	Professori Keijo Heljanko		

Regressiotestaus on testauksen muoto, jonka tarkoituksena on varmistaa että ohjelman olemassa olevat testit eivät löydä virheitä muokatusta ohjelmasta. Regressiotestaus suoritetaan yleensä jokaisen ohjelman muutoksen jälkeen, ja sen suoritus voi viedä paljon prosessointiaikaa. Regressiotestien valinta on prosessi jossa ohjelman kaikkien testien joukosta valitaan muutoksen kannalta oleellinen testien alijoukko. Valinnan tavoitteena on pienentää testien määrää ja näin vähentää testien suoritukseen kuluvaa aikaa. Turvalliset regressiotestien valintamenetelmät ovat menetelmiä jossa voidaan todistaa että valitsemattomat testit eivät olisi voineet löytää virheitä, ja täten ne voidaan jättää suorittamatta.

Tutkijat ovat kehitelleet useita eri menetelmiä turvalliseen ja epäturvalliseen regressiotestien valintaan. Useat menetelmistä tarvitsevat ohjelmien ohjausvuokaavion tai vastaavaa informaatiota, jota voidaan laskea ohjelmien käännöksen yhteydessä. Tämän vuoksi menetelmät eivät ole yhteensopivia dynaamisesti tyypitettyjen tulkattujen ohjelmointikielten kanssa, joissa tätä informaatiota ei ole saatavilla.

Tämä työ esittelee testien kattavuuteen perustuvan menetelmän regressiotestien valintaan, jota voidaan käyttää tulkattujen ohjelmointikielien kanssa. Esitelty menetelmä ei tarvitse muutoksia testattavan ohjelman ohjelmakoodiin. Esitellyn menetelmän testien valinnan tarkkuutta testattiin keskikokoisella verkkosovelluksella, ja tulokset olivat osittain ristiriitaisia. Testien valintamenetelmä onnistui valitsemaan pienen testijoukon noin puolessa testitilanteita. Lopuissa testitilanteista menetelmä joutui suorittamaan kaikki testijoukon testit. Menetelmän käyttämisen havaittiin kuitenkin hidastavan valittujen testien suoritusaikaa merkittävästi.

Asiasanat:	testaus, regressiotestien valinta, testien kattavuus
Kieli:	Englanti

Contents

1 Introduction **6**
1.1 Problem statement 8
1.2 Structure of the Thesis 9

2 Background **10**
2.1 Software development process 10
2.2 Testing in software development 11
2.3 Test prioritization 13
2.3.1 Coverage based priorization heuristics 13
2.3.2 Source code complexity based priorization heuristics . . 14
2.3.3 Source code change set based priorization heuristics . . 14
2.3.4 Past fault detection capabilities 15
2.4 Test minimization 15
2.5 Test selection methods 16
2.5.1 Unsafe test selection 17
2.5.2 Safe test selection 18
2.5.3 Issues with dynamically typed programming languages 19
2.6 Other methods of improving the performance of test suite . . 20

3 Implementation **23**
3.1 Idea behind coverage based test selection 23
3.2 Overview 25
3.3 Extracting dependent files 26
3.4 Extracting execution paths 29
3.5 Transforming execution path to scopes 31
3.6 Mapping change set to code scopes 32
3.7 Limitations 34

4 Results **37**
4.1 Tested application 37
4.2 Test arrangements 38

4.3 Test deselection results 39
4.4 Test performance results 40
4.5 Discussion 41

5 Conclusions **44**
5.1 Future work 45

Chapter 1

Introduction

Testing is widely used method for identifying and reducing the number of faults in software. Testing consist of tests that contain some code segments that define how the tested software should behave. The test consist of construction phase where the initial software state is being initialised. After the construction the test will execute the testable section and after that these is the phase where the software state is compared to expected value. Usually there are many tests that require the same software initial state. To reduce the amount of initialisation code, the construction code can be split into fixtures. A fixture is a code segment that contains code that initialises some part of software state. Usually when executing tests, each test will execute each of their required test fixtures. In some cases it is possible to share fixture between many tests, and so reduce the time that is taken by fixture construction in test execution.

Normal software can be seen as taking some input, processing the input in some way, and finally returning some output while at the same time terminating the program execution. With web applications the input is the request that is usually sent by a browser and the output is response sent back to the requester. Unlike normal software, web application usually do not terminate after processing one request, but rather keep serving incoming requests until being terminated by some other program. Usually with web servers all the server state is stored to a database or multiple databases, and the actual web-server application does not store any state information. In some cases the web-server may store some intermediate results to some calculations in its internal caches, but those results should not have an effect to the responses it sends to requests.

With web applications the request path usually determine what processing steps the web application should take when processing that request. For example path `/` might show the landing page where as path `/contact` might

return page with contact information. Therefore each web application can be seen to contain multiple different sub-applications and the request path determines which one of those application gets executed.

Knowing that each test usually tests just the result of a single function, it is likely that a change to certain sub-application does not have any effect on tests that will never execute that code. Therefore we could annotate each test case with information about which sub-application this test case expects to use. This would allow us to only run a subset of test cases when some sub-application gets changed. Creating such annotations would take considerate amount of time and possible mistakes could either increase the time that test suite execution takes, or in the other case not run all the test cases when there are changes to the tested code. Therefore it would be preferred to have some automatic program handle the annotation of test cases so that the risky manual work is avoided.

With dynamic programming languages the code execution flow can become really complex. Determining code execution flow from given code segment if often impossible, due to the lack of static typing. Therefore each given input can be of any type, and accessing any member of an object can cause arbitrary side effects. Also most web application frameworks usually use lots of programming language features while trying to make using the framework as pleasant and effective as possible. Disallowing some of the language features to make the execution flow easier to reason about would hinder future web application development and prevent the use of this method on existing programs. One possible way of figuring out the dependent code sections for a test is to measure the code execution coverage during the test run. From the execution coverage we can determine which functions are not executed during the execution of the test. When we are recording the test execution trace we must also include all the code that is executed when the test fixtures are being executed.

In Python source code each function consist of two parts, name binding operation and the function body. When we are tracking the code execution coverage, we can usually see that all the code is only imported once as the name binding operations are not executed during the test execution. The code import phase happens only once during the test suite execution when the source code files are imported. Changes to function name bindings can cause wild changes on anywhere in the code base, as we are unable to keep track of code reflection in our code base.

1.1 Problem statement

As the web platform has formed into universal web service platform working in all devices, many companies have started building more and more web applications. As the actual application is running on web servers, has the time and effort required to bring out new features and bug fixes to the users of the services decreased. As the feature release pipeline has gotten shorter, the specific quality assurance steps have been mostly replaced by continuous integration services running the application's test suite. These test suites try to find new regressions as existing bugs are fixed and new features are being added to the application. As the number of features in the application grows, the test suite containing tests for these features will also grow. Running large test suite will take considerable amount of time and this hinders the application development speed. The developers may offload the test suite execution to Continuous Integration (CI) server and continue working on some other bug fix or new features while the test suite is running, but this introduces a new context switch to the work flow. As the test suite keeps increasing there may be need to parallelize the test suite execution to several CI servers and this will slowly make regression testing more expensive.

As the hardware performance has increased, the relative expense of software engineering has increased. This has increased the usage of dynamic programming languages like Python and Ruby in web application development. Dynamically typed programming languages allow more expressive source code compared to statically typed languages. At the same time the testing effort need to be extended to offset the missing compilation step, that is likely to catch simple faults in programming code. These dynamically typed programming languages can't take advantage of compilation to machine code without special arrangements, so the performance is also worse when compared to compiled languages.

Most medium to large web applications contain hundreds of different endpoints. Each endpoint is a single sub-application that gets executed when request's path matches the endpoint's path pattern. Each endpoint is responsible of doing certain actions and returning certain page. As most bug fixes and new features only modify the source code ran by very few endpoints, running all the tests for each endpoint increases the test suite runtime unnecessarily. This thesis presents a way of intelligently storing test execution trace and dependency information and running only the test cases where the execution trace contains modifications.

Decreasing the test suite runtime can increase development efficiency, as there is less waiting for tests to pass. Even with slow test suites the

test selection method may offer significant runtime reductions allowing the developer to run the test suite on their own machine instead of waiting until the CI service has ran the tests. By being able to run the subset of tests that is covering the current change set the developer can run the subset of tests and safely ignore all other test cases. This may bring to test result feedback loop to fraction of it previous length. On CI services the faster test execution will save money, as less hardware is needed to run the same test suites. Likewise by not running some test cases the CI service can return test suite result faster.

1.2 Structure of the Thesis

This thesis consist of Chapters: Introduction, Background, Implementation and Results. In the introduction chapter we explain what this thesis is all about and why it is important. In the background chapter we will explain regression testing in more detail and go through different methods designed to make regression testing faster. Some test selection methods are "safe" as in they will never fail to detect faults, and other methods are "unsafe" as in they can fail to detect some faults. As explained in the introduction chapter we are more interested in safe test selection methods, as then out method will always detect all the faults that the normal tests suite execution would. In the background chapter we also explain why some of the explained methods can not be used with dynamic programming languages.

In Chapter 3 we explain in details our proposal to make regression testing faster with coverage based safe tests selection method. Our goal is to mainly target web applications written in Python programming language, but other Python applications should also be able to use our method without any changes. The chapter also contains discussion about limitations of our test selection method.

In Chapter 4 we test our implementation with medium sized proprietary web application. We utilize the tested application's source code repository in order to test how well the implementation would have performed with in actual code changes that have been implemented to the application in the past.

In the Chapter 5 we conclude our thesis and discuss the limitations and future research possibilities.

Chapter 2

Background

2.1 Software development process

The simplified process of developing a hotfix or a new feature to a application is shown in Figure 2.1. A hotfix is a small modification that fixes previous conflict between the implementation and the specification, or changes some small piece in the specification. The process starts from specification that defines how the application should behave. The specification might be for a single feature, or modification to existing feature. The developer implements the specification to the application by modifying and extending the existing application source code. After the developer has made the changes to the application source code, the next step is to write a set of new tests that test that the new feature works as intended. If the specification is about a hotfix the developer may write a regression test that makes sure that the application behaves as the specification defines. The testing phase also consist of running portion of tests that are related to the changed source code section. Usually the developer chooses this test set manually without any help from automatic tools. If any of the chosen tests finds a fault in the application the developer needs to return back to implementation phase to make sure that the tests pass. After all the chosen tests pass without finding any faults the changed application can be moved to regression testing phase. In regression testing the whole application test suite is executed to make sure that none of the tests find any faults in the application. If faults are found, the developer needs to return to implementation phase to fix the found faults in the application, or to change the tests to behave as the new specification defines. After the regression test phase has been passed, the changed application is moved to code review phase. In this phase some other developer makes sure that the changes to the application source code behave as the specification expects

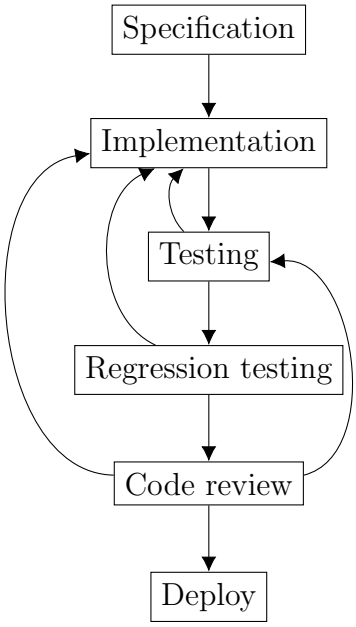

Figure 2.1: High level overview of the development process

and that the changes are easy to understand. If this is not the case, the development phase is returned to implementation phase. It is also checked that the new feature or hotfix is tested enough. In case the testing is lacking the phase is returned to testing section. After the code review is done, the new feature or hotfix is ready to be deployed to the production use.

2.2 Testing in software development

Software testing is a process with the goal of finding defects in the tested program [19]. A defect is any kind of incorrect behaviour done by the tested program. Incorrect behaviour can range from function returning invalid value to taking too long to calculate the result. Incorrect behaviour can also be the action of calling some unwanted function or forgetting to call some other function. The incorrect behaviour for each test is specified in the test. Testing is considered to take significant portion of the total time of typical software project.

Testing as a process consist of running a test suite. A test suite is a collection of test cases. Test case consist of program input state and some code sequence to be executed. In the rest of this paper we are going to assume that the tested code sequence is unable to access information outside the input state. With this assumption each test case can be seen as deterministic

state machine, as the tested code is unable to get any non-deterministic data in that it could use to change the execution trace between different execution times. We are also going to assume that there is always just one test executor process, and so rule out the option of getting non-deterministic execution from different process execution orderings. The test's code sequence will return an boolean representing whether the test case found a defect. The tests can be seen as small subprograms that execute some part of the tested program and make sure it behaves in the expected manner on one single input case.

Testing is an important part of software development as it increases confidence to the expected behaviour of the application. Usually a test suite consisting of multiple test cases is created when the application is created. Regression testing means running modified software against the existing test suite. The purpose of regression testing is to find new defects that the newly added modifications introduced to the the application. As the application development is continued, new test cases are created to make sure that the new features or bug fixes are working. These new test cases are added to the test suite so that all the following regression tests runs check that the there hasn't been regressions in these features or bug fixes. During development some of the existing test cases might become obsolete, meaning that the test case is no longer matching the expected behaviour. The obsolete tests cases are removed from the test suite so that the test suite consist of test testing the expected behaviour of the program.

Mathematically testing can be explained as follows: Let P be our program, P' our modified program and T our test suite for P. Regression testing is testing P' with T. Because with large tests sets the testing is time consuming, some algorithms have been developed to test P' against some smaller subset of T. This is called test selection. Some algorithms have also been developed to make fault detection faster by ordering the tests according to some external factor. Test coverage and previous test fault detection capabilities have been some of the proposed factors [26].

After a modification have been made the the application, the tests can be divided into four groups: Reusable, retestable, obsolete and new [30]. Reusable tests are tests that are still valid after the change, and where the modification did not have any effect on the execution path. As there are no changes in the execution path, these test do not need to be retested during regression test. Retestable tests cases are test cases where the test execution trace contains modifications. These tests need to be tested during regression test to make sure that the modified application does not contain faults. Obsolete test are tests that are no longer needed after the application modification. New tests cases are tests cases that were written with the

application modification to make sure the modification works as intended.

With complex software and extensive test suite the execution regression tests tend to take considerate amount of time. Finding faults in regression suite faster would give major benefits to the speed of software development, as the developers would not need to wait so long to get the test results. There are several techniques that aim to make detection of failures in test suite faster, including test selection, test prioritization and test suite minimization.

2.3 Test prioritization

In some cases the developers want to have the information whether any test in the test suite will find a fault or not as soon as possible. Especially with long running test suites getting information about the failure in the beginning of the test suite may let developers know about the issue many hours earlier compared to a test failure that happens closer to the end of the test suite. Test prioritization is a technique that alters the order in which the tests are executed. Its goal is to make the test suite execute fault revealing test cases as soon as possible. If an fault revealing test case is found, the developer can be notified before all of the test cases have been executed. Optionally the test suite execution can be terminated after the first fault revealing test case has been found. There are multiple different prioritization techniques with different performance behaviours [17, 26].

Test priorization techniques are similar to test selection techniques. The only major difference in the methods is that the test selection algorithms have some end condition that specifies when rest of the test can be left unselected. Every test selection technique can be turned into test priorization technique by first running the tests that the test selection algorithm selected, and then running the unselected tests. Every test priorization technique can be turned into test selection technique by adding it some end condition. For example one such condition could be that select the first half of the tests ordered by the priorization technique and leave the other half unselected.

Test priorization can order tests based on many different techniques. Usually the priorization is done by some simple heuristic or combination of heuristics. The most used heuristics are covered in following subsections.

2.3.1 Coverage based priorization heuristics

Coverage based priorization heuristics are quite common. In coverage based heuristics the tests are ordered by the number of code statements that are covered by the execution of each test. Similarly the test ordering can be

based on how many additional statements get covered by the test suite when the test is executed. Both of these techniques can also be calculated on function level, where instead of measuring statement coverage, we measure function coverage. Function is considered covered if it gets called during the test execution. It is impossible to extract the tests' statement or function coverage after arbitrary changes have been applied to the application without running the tests, but we may assume that the coverage is really close to the coverage of the previous version, and use the coverage information from that test execution. One other alteration to this method is to greedily order the tests based on how much new coverage value the tests adds.

Using the test coverage as an input for priorization heuristics is easy as the coverage information is quite often already available. The reason for the availability is that various testing standards have mandates for acceptable coverage values [17], so the coverage is already being measured.

2.3.2 Source code complexity based priorization heuristics

In addition to the coverage based priorization techniques there are also techniques that try to order tests based on arbitrary probability that the test fill find fault. The probability can be approximated by mutation analysis where application source code statements or functions are being exposed to modifications and then the existing tests are given scores based on how many faults the test found in the mutated application [7]. The idea behind this method is that by using mutation analysis we can calculate approximation of how likely each test will find mutations in the source code covered by each test. One other alteration to this method is to greedily order the tests based on how much probability score value each test adds.

2.3.3 Source code change set based priorization heuristics

There are also approximation heuristics that calculate the test's fault finding probability by comparing the code changes and giving each function fault index value based on the complexity of changes made to the functions. The test's fault probability is calculated by summing the executed functions' fault indexes and using that sum to order the tests [7]. Simpler version for fault index calculation is to simply use the number of added, removed, or modified source code lines in a function and use that number as the approximation of how likely that function is to contain new fault [8]. The major difference

when compared to the other methods mentioned above, is that in the source code change set based priorization heuristics the test heuristic result depends on what changes the tested version has. This sounds reasonable as if the change set only modifies small portion of the tested application, the faults revealed by the regression testing are most likely to be revealed by the tests with coverage on the change set. One other alteration to this method is to greedily order the tests based on how much new fault index value each test adds.

There are also heuristics that take the compiled application binaries as an input instead of source codes [27]. In this case the solution uses basic block granularity to detect which tests cover which sections of the application. Also application changes are detected at basic block granularity. A basic block is maximal group of statements with single entry and single exit points. This solution is shown to work with large software applications [27]. One positive aspect of using compiled binary executables as a input to the heuristics is that it removes incorrect priority changes caused by renaming of variables in source code. Any heuristics that takes compiled binary application as an input can not be used with application where the application source code is executed with interpreter. The issue is that the traced coverage or basic blocks are for the interpreter code and not the actual application source code. The application source code is just data that the interpreter processes and therefore the solution would be unable to detect any changes to the application code.

2.3.4 Past fault detection capabilities

One other way of approximating test's fault finding probability is to use the test's historical fault detection performance from the previous test suite runs [15]. This method assumes that tests that have detected faults recently are more likely to find faults than the tests that have not found any faults recently. The method is designed for situations where running the full test suite is not simply feasible, and only a portion of it can be run during testing. The method will prefer to run tests that have not been run recently. This will cause the method to cycle through all tests over a number or test suite executions.

2.4 Test minimization

Test minimization is a technique to reduce the number of test in a test suite. The idea behind test minimization is that there are certain set of requirements

that the application should satisfy [30]. With the assumption that each test can test one or more requirements, it is possible to discard tests of which would test some already tested requirements. There are multiple basis on which the minimization can be based, coverage is one often used. It is easy to find examples where the assumption behind test minimization does not hold. In Listing 2.1 we have example where function `multiply` should be doing multiplication operation, but is actually doing addition. As the coverage of test `test_multiply_2_2` already covers everything that `test_multiply_2_3` covers, the test minimization techniques could discard the fault revealing test `test_multiply_2_3`.

In real life situations it is easy to see similar cases. Lets assume that we have a webpage to where a user can post new orders. When the page receives new order, it should store the order details to database and send the user confirmation email about the order. Lets also assume that there are two separate integration tests; `test_order_email_content` for the sent email's content and `test_order_saved_to_database` to test database storage process. In this case both of these tests may have exactly the coverage in the application source code, and one of them may be discarded. The assumptions that the test minimization is based on may not apply for such applications where the tests are written by developers, and such using it to reduce test suite execution time might not be suitable solution in these cases. There is also research that suggests that the test suite minimization can significantly compromise the fault-detection capabilities of test suites [25].

Listing 2.1: Minimal example where test minimization fails

```
def multiply(a, b):
    return a + b

def test_multiply_2_2():
    assert multiply(2, 2) == 4

def test_multiply_2_3():
    assert multiply(2, 3) == 6
```

2.5 Test selection methods

A test in a test suite is modification-traversing if there is code modification, insertion or removal within the test's execution path [22]. Test selection techniques that always select all modification-traversing tests are called safe test selection techniques.

2.5.1 Unsafe test selection

There are also techniques where redundant test runs are avoided [21]. Random test selection and test minimization are some examples of these techniques. Both of these methods will reduce the number of test executed, but also the fault detection capabilities of the test suite [11, 14].

In data flow based methods program slicing is used to create definition-use pairs for all variables [12, 30]. The test selection works by selecting all tests that cover any changes in definition-use pairs between the different program versions. Because statement that doesn't affect function output is not visible in definition-use pairs, it is possible to add certain type of statements to a function without data flow based test selection methods noticing it [30]. Therefore the method is not safe test selection method. Also with data flow based methods it is undefined how the control flow of the program is handled when the programming language allows changes to control flow by throwing exceptions.

In dynamic slicing the execution trace of each test is calculated and only the statements that affect the test output are marked as covered [1]. By only running the tests that had changes in those covered rows, we can ignore changes that do not affect the test output. The dynamic slicing by itself is unsafe because conditional jumps that were not taken do not directly affect the test output. Calculating the needed dependencies between data and control statements is more complicated on dynamic languages like Python, but it is possible [6]. Relevant slice is a superset of dynamic slice that also contains statements that could have affected the test output had they evaluated differently [1]. Even when relevant slice is used to select the test the technique is not safe. The technique is not safe as it is possible to add new statements that do not create new variable or change any existing variables. These statements can be modification revealing, but they are not visible in relevant slices [30]. Therefore the tests that would execute these added statements are not selected.

In firewall based methods a figurative firewall is build around the modules with changes in the code or specification [16]. The firewall will be constructed using control-flow and data-flow information [29]. The firewall will limit the modules to which there is need to run integration tests. The method is built around several assumptions that may not be easy to fulfil. For example the method requires that the program call graph needs to be acyclic, meaning that there is no recursion [16]. Also the method imposes severe restrictions to the usage of pointers in applications. As in Python variables store references to objects and references are technically pointers without arithmetics, it can be seen that the referenced firewall based methods are unsuitable to use in

any non-trivial Python application.

One other option is also to use manual test selection, where some experts decide which tests should be selected. Manual test selection is considered ineffective and may be unreliable for larger programs [4]. The problem is that with even with moderately sized programs finding the relevant test cases for some code change is hard, as the program internals may be unclear to the person doing the test selection process.

2.5.2 Safe test selection

Safe test selection is a case of test selection where it can be proven that the tests that were not selected can not fail [23]. One safe test selection technique is based on Control Flow Graphs (CFG) [23]. CFG represents all possible control flow paths that can be taken when the application is executed. With CFG methods a CFG is constructed of the application source code. By tracking which tests traverse which CFG edges we get to know which CFG nodes and edges are visited by each test. By mapping the code changes to the CFG it can be seen which test traverse to CFG nodes with changes. All the tests than do not traverse to changed nodes can be left unexecuted, as there cannot be any changes to the result of those tests. Construction of CFG for an application may not be easy. In one implementation for C-language the implementation failed to analyze 15% of the tested application's procedures [23]. In another implementation for C++ -language the whole analyzing step was replaced with simulation because suitable analyzer application was not available [24].

TestTube [5] is an solution for safe test selection for application written in C language. The TestTube works by modifying the application source code so that the application execution will produce list of all functions that were executed. Then with static analysis tools the information about used variables is added to the function lists, and all this information is stored to a database. During the test selection phase the static analysis is redone and the changes are seen by comparing the results with the previous static analysis. Based on the changes this solution will select the tests to which the change set between the versions could have affected. The static analysis steps most likely can not be implemented to Python programming language because of its object-oriented features and reflection capabilities. One such case is seen in Figure 3.4 in Section 3.7. The addition of function `add_two` will change the output of `test_unknown_function` because the reflection exposes the new function. Making the static analysis understand all similar situations robustly is a hard issue. The referenced research didn't include any mentions of testing common refactoring changes: adding, removing and

renaming functions.

Pythia is another test selection tool for C programs [28]. Pythia works by modifying the application source code so that the application execution will produce basic block execution trace for each test. A basic block is maximal group of source code statements with single entry and single exit points. Both the basic blocks and their execution trace are stored for the test selection phase. In the test selection phase the source code files of the new and the old version are compared with UNIX tool diff and based on the differences all the modification-traversing test cases can be selected. Converting the approach taken by Pythia to Python language might not be practical, as the concept of basic block does not suite Python. As the Python code is interpreted it is not easy to find any basic blocks.

There have been some surveys comparing different test selection methods [9], but none of the listed methods are directly compatible for dynamic programming languages.

2.5.3 Issues with dynamically typed programming languages

Most safe test selection algorithms require code flow graph or similar for the application code [9]. As with all dynamically typed languages with first-class functions, getting static code flow graph is not possible [3]. There have been a few ideas how to get approximation of it, but so far no reliable method for extracting it have been found [10]. It is possible to create code flow graphs for portions of the application code when the types of variables are fixed [3], but that is not robust enough for safe tests selection. For example the code `a + b` code flow graph can be calculated when both variables are of type integer. If the variable `a` happens to be some other type with custom `__add__` method, the calculation of code flow graph would get complicated. As the types of variables are not fixed, it might not always be feasible to calculate the possible types for each variable.

The point of analyzing control and data flow graphs is to get more precision to the test selection. As seen on the previous section getting the analysis to work robustly is not easy for even for statically typed languages. With dynamically typed programming languages the analysis gets even more complicated. The dynamic typing preventing getting variable type information, first-class functions and classes causing issues with graphing dependencies, and the possibility to mutate data structures during execution are some of the issues that prevent the construction of graph of data and control dependencies [6]. In the presented implementation we will omit all complicated

analysis steps, and simply fall back to rerun all case on non-trivial changes to application source code.

The positive side of interpreted languages is that each source code line gets directly executed as it is without any compilation steps. This makes it easier to track which source code lines have been executed. For example with compiled languages the compiler may replace function calls with the function body to avoid the function call overhead [13]. There are several different optimizations that the compilers can do, and these will make it harder to get information about the executed source code statements. It may be possible to disable these optimizations during the test suite execution, but doing that may also negatively affect the amount of time it takes to execute the test suite. With interpreted languages it may be possible to get information about the executed source code statements during the application execution. The presented implementation is going do exactly that, and the method is explained in more detail in Section 3.4.

2.6 Other methods of improving the performance of test suite

In this section the word performance only means the execution speed of the test suite, and not the fault detection ability of the suite. There are multiple other ways of improving the testing performance, such as:

- Replacing integration tests with faster unit tests.
- Running tests in parallel on many computers.
- Mocking slow parts of the tested application.
- Splitting the web application to microservices.

Because in the reference implementation of Python programming language the compiled bytecode is interpreted, the performance is worse when compared to compiled languages. On the other hand the language is more expressive than most compiled programming languages, which makes it suitable for web development, where coding efficiency is important. Because of the slower execution speed and good coding efficiency it is possible that the performance issues in test suite are more likely to hit Python projects compared to other more static programming languages.

The most trivial way of making the test suite more performant is to run less code in test suite. Unit tests test only small portions of the application code. With integration tests a lot bigger portion of the application source

code is executed. By designing the application in a way that makes it easy to test portions of it using unit tests improves the tests suite runtime when compared to test suite where all those tests are integration tests. The process of turning some existing integration tests to unit tests is likely to take lots of development time and is so expensive. It is also possible that some of the changes will reduce the fault detention abilities of the test suite, as fewer code lines gets executed.

To make test suite execute faster, one might want to take advantage of the parallelism in test execution. Because the Python interpreter cannot support more than one simultaneous thread execution [2], threading solutions are not feasible in situations where the execution is CPU limited. If the test execution does lots of IO-operations the threading solution may be able to speed up the test execution. The one simultaneous thread execution limitation can be worked around by dividing the test suite to multiple smaller subsets, and then running each subset in its own process. After each test subset has been executed the results can be combined and returned as if all the tests were run in sequentially. The subsets can be created so that the total test execution for the tests within the subset are close to one another. This way all the test suite subsets complete close to one another and the parallel execution is maximised. The different subsets can be executed on a single or multiple computers. On a single computer the amount of memory and processor cores will limit the number of parallel test executions. Running tests in parallel on many computers increases the resource need and therefore costs. It also makes testing infrastructure a bit more complicated, as the different computers need to distribute the test subsets and collect the results.

Another way of speeding up test execution is to replace the slow sections of the code with mocks, that return some predetermined value without actually executing the time wise expensive operation. The slow section might be calling some external other application, doing something IO-wise expensive or simply some CPU-wise expensive operation. Some examples for such actions might be expensive database operations and repeated cryptography operations. Changing the test suite to use more mocking takes development time. It might also reduce the fault detection ability of the test suite if the mock doesn't behave as the mocked code section does.

With micro-service approach the application is split into several smaller independent micro-service applications that each handle just small portion of the original monolithic applications tasks [20]. Each micro-service has its own test suite, and for changes that do not modify the micro-services interface only the tests inside that micro-services test suite need to be considered when running regression tests. Changes to micro-services interface should be rare when the micro-service is correctly designed from the start. Converting

monolithic application to use micro-service approach increases the development overhead and makes the system architecture more complex. Therefore the decision whether to go with micro-service based approach should not solely rely on whether it would make testing more efficient.

Chapter 3

Implementation

In this section we are going to present a method for safe test selection that will select all the fault revealing tests cases that rerunning all the tests would. This way the test suite execution runtime can be reduced significantly. The Section 3.1 contains mathematical explanation of the idea behind the implementation and all the assumptions that must hold for it to work as expected. This is followed by Section 3.2 where overview of the implementation details are given. The Sections 3.3, 3.4 and 3.6 will explain the different parts of the implementation in more detail. This chapter's final Section 3.7 will explain some of the limitations that this test selection method has.

3.1 Idea behind coverage based test selection

Let P be our program, P' our modified program and T our test suite for P. Let R be any single test case in T. Assuming that the program code and test setup is deterministic, and that the test environment behaves deterministically, the execution of test R will always take the same execution path covering statements $S^{test} + S^{program}$. When the change set containing all changes between P and P' does not contain changes to any statements in $S^{program}$, with the assumption that $S^{test} + S^{program}$ does not access the source code or any other form of any of the statements changed between P and P', the test R execution path when run with P' must deterministically take the same path as it did when tested against P. Therefore the result of test R must be identical on both programs P and P'. Assuming that test case R did not find any faults in P it can seen that R can not find any faults in P', and therefore there is no need to run that test.

This proof relies on three assumptions that must hold. First is that the application source code must be deterministic. If the program is non-

deterministic, the coverage recorded in analysis phase may not contain some reachable statements. This allows the test selection phase to leave these test out of the selection, as there exists an execution path that will result in the test passing. With non-deterministic code the presented method is as good as re-run all, as rerunning the test may not find the fault either.

The second assumption made in the proof is that the environment must behave deterministically. The main limitation here is that we are limited to testing single-threaded applications, as otherwise the different interleavings in threading would make the program behave non-deterministically. This also forces the user to keep the same interpreter and prevents changes to the versions of installed libraries. By requiring deterministic environment we are also ignoring all possible issues that may come from the environment, such as execution timing differences between test runs. In our test application the test setup and the application code will read the contents of files from filesystem. To validate the assumption that the test environment will return the same content between the test runs we need to manually keep track of which tests read which files, and when the contents of those files changes. This logic is explained in detail in Section 3.3.

The third assumption made in the proof is that the program will not access any unexecuted functions source code. There are several different ways to access the source of a python function. Python standard library offers `dis` module[1] that can extract function's bytecode representation. Raw source code version can also be extracted using `inspect` module[2]. The presented test selection method assumes that any such way of accessing unexecuted functions code is not used. It is assumed that the extraction of bytecode or source code versions of a function is not used that commonly in application production code.

An example application is shown in Listing 3.1. If the change set only contains modifications inside function `func_a`, then the execution coverage for `test_func_b` still stays the same. Therefore changes to `func_a` do not force use to run test `test_func_b`. On the other hand if there are any modifications to `add_one` or in any of the functions' name bindings, all tests need to be re-run.

[1] https://docs.python.org/3/library/dis.html
[2] https://docs.python.org/3/library/inspect.html

Listing 3.1: Source code of simple example case

```
def add_one(i):
    return i + 1

def func_a():
    return add_one(1)

def func_b():
    return add_one(2)

def test_func_a():
    assert func_a() == 2

def test_func_b():
    assert func_b() == 3
```

3.2 Overview

The safe test selection method has two phases: Analysis and test selection. In analysis phase all the tests in a test suite are executed while tracing the executed source code. All the executed source code statements during a single test is referred to as the test coverage. During analysis we also track all file access operations for each test. All tests that are accessing files are marked as dependent on those accessed files. The test selection phase is done after analysis phase when we want to test that all the regression tests in the test suite still pass. At the selection phase we have dependency and coverage information from the analysis phase as well as a change set containing all changes done to all files between the analysis phase and test selection phase. We can use file dependency map and execution coverage measurements in the test selection phase to only select those tests that have modifications in either executed source code, or in the dependent files.

Our test selection method's analysis works by using each test's execution coverage to build a list of distinct scopes that are visited by the test. A scope in this context is one or more source code statements depending on how precise granularity is wanted. The smallest and most precise granularity is program statements. Other higher levels of granularity are function, class and module. Most safe test selection methods use statement level granularity [9]. More smaller granularity levels can deselect more tests than larger granularity, as there is more information about the test execution coverage.

However it also increases the processing and storage requirements for the analysis phase. In this solution we have decided to use function level granularity, as it offers good balance between precision and performance. With the assumption that the tested applications functions do not contain excessive number of branching, the statement level granularity would not help the implementation to deselect significantly more tests. As later seen in the Section 4.3 the achieved deselection performance with function level granularity is not an issue with the tested application.

The application source code and the test suite are expected to be stored in a Git version control system repository. The source code repository consist of successive commits, each of which stores a change set and the the information about the preceding commits [18]. Each commit can be identified using an unique hash of its contents.

The high level overview of the presented method is shown in Figure 3.1. In the analysis phase the application test suite is run while recording the test coverage for each test separately. Then using the coverage data each covered source code line is mapped to a scope. A list of tuples containing test identifier and scope information is stored into a database, where it can be easily queried during selection phase. Each tuple also contains hash of the current commit in source code repository. This hash is used to calculate what is the actual change set during the selection phase. In the selection phase all the changes in the change set are mapped to the source code, and code scopes are being calculated for the changes. These change set scopes tell us all the scopes that have been modified. The actual test selection is simply selecting all the tests that have same scopes as the change set. In case the change set makes modifications to the code outside functions, all the tests need to be re-run. Creation or removal of global variables or classes are some examples of such modifications. We call these modifications code structure modifications.

3.3 Extracting dependent files

File access operation is an operation where the full or partial content or other attributes are accessed from a file in a file system. File access operation could also be operation where the test or tested code would write to some file in filesystem, but assuming that the tests don't communicate with one other file write operations can't change the result of any other tests. Therefore write operations can be ignored. Files that are accessed during the test execution are called dependent files. These files can be accessed by the tested application or the test code. With a web application the accessed files may

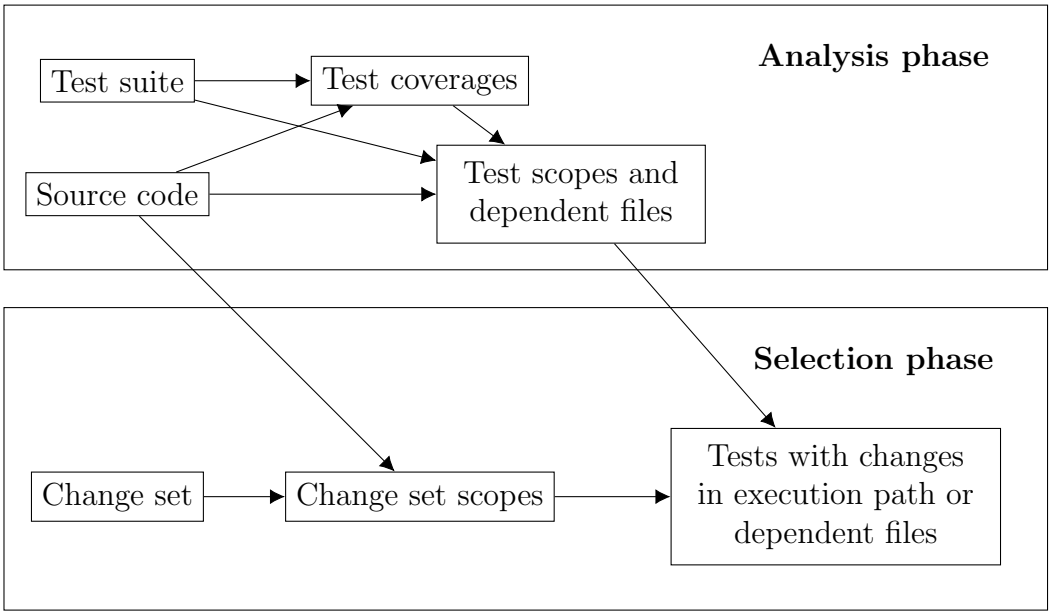

Figure 3.1: High level overview of the process

include static translation database and templates. Templates are files that are used to construct HTML responses to request. Templates are usually mix of HTML markup code interleaved with application source code that handles all the dynamic parts of the page.

File access operation can target a file that is part of the repository containing the applications source code and test suite. In these cases the file access is interesting, as any source code change set may contain changes to that file. When the accessed file is not stored in the source code repository, the application change sets cannot contain changes to this file. Therefore the application source code change sets do not contain changes to these files, and these file access operations can be ignored. The exception to this is if the targeted file that is not in the repository is later added to the repository. In any case we can't track changes to files that are not stored to the repository, so we are going to ignore such changes. We will assume that all accessed files outside the repository are never changed, or the contents are determined by some other file that is stored to the repository.

One common case for accessing files outside the application repository is a case where application dependency library is loading some static data files from its installation folder. With Python applications the dependency libraries are usually resolved and installed by a program called pip. Pip supports specifying the exact versions of the dependency libraries as well as hashes of the downloaded libraries. With the assumption that the application

dependency libraries are always installed deterministically, it is reasonable to assume that the static data files of libraries are not changed. All the dependency libraries are usually defined in a file that is stored in the application repository. That file can also specify the exact version of each library that makes sure that dependency libraries will not change when the dependency resolution is done later. The same file can also include hashes for the downloaded libraries to make sure that no one can change the content of libraries while keeping the same version number. Therefore it can be assumed that any change to dependency library needs to cause a change to the file containing the dependency library's version and hash. A change to that file will result in re-running all tests again, so the changes to dependency libraries are detected automatically.

Another limitation to file access operations tracking are file listing operations. The tested application code or test code may ask the operating system to list all the files in certain folder. We will simply assume that either the result of such operation is always the same, or that the difference in the operation result will not have any effect to the code runtime. It is assumed that these cases are rare, and any such operations are not tracked.

With the Python testing environment we can track file access operations by overwriting the file access functions in Python language's standard library during the test initialisation. The overwritten file access functions will keep track of accessed files, but will otherwise behave exactly the same way as the original file access function. The presented implementation will only hook `open`-function, as it was used for all file access operations in the tested application. There are other functions in Python standard library that may be used to access file contents or attributes in the file system, but those are ignored in the implementation. The tests can also access file contents by calling external libraries or programs, but those file access operations are also ignored.

All file access operations done during tests initialisation phase in fixture construction are marked to affect all tests that use that fixture. This is demonstrated in Listing 3.2. The fixture `sample_line` is reading the contents of file `sample_file4`, so as test `test_func` uses that fixture, it gets dependent on that file. Similarly the test function `test_func2` directly opens the same file, so that test is also dependent on that file. Both test function will later on call function `text_found_from_sample_files`, which open either just file `sample_file1` if the condition on line 6 holds, or both `sample_file1` and `sample_file2` if it doesn't hold on the first round on loop on line 4-7. The files opened by the function are also marked as dependency files to both test functions.

Listing 3.2: Source code of example demonstrating different file access operations

```
from pytest import fixture

def text_found_from_sample_files(text):
    for filename in ['sample_file1', 'sample_file2']:
        content = open(filename).read()
        if text in content:
            return True
    return False

@fixture
def sample_line():
    return open('sample_file4').readlines()[0]

def test_func(sample_line):
    assert text_found_from_sample_files(sample_line)

def test_func2():
    sample_line = open('sample_file4').readlines()[0]
    assert text_found_from_sample_files(sample_line)
```

3.4 Extracting execution paths

In order to know which source code lines are executed by which tests we need to be able to track the execution path of the each individual test. As any test fixture construction phase can execute application functions, we must also track the execution path during fixture construction. The execution path taken by fixture construction is shared to all tests using that fixture. Also the execution path taken during code import phase is shared between all tests. The execution path during import phase usually contains just the few used custom decorator functions.

The executed source code lines can be divided into three groups based on the source code file location: Application source code, Test suite source code and dependency library source code. We are only tracking the execution path in application source code. The dependency libraries are not usually stored in the application repository, so we cannot calculate the change set during test selection phase. Also it can be assumed that modifications to dependencies happen relatively rarely compared to application source code.

The third reason to exclude dependency libraries execution paths is performance. Tracking the execution path causes some processing overhead and storing the results also takes more space.

The execution path inside test suite source code can also be ignored, as we can simple rerun all tests in any files that were modified by the change set during test selection phase. This works with the assumption that tests do not import code from other test suite test files, and changes to general test suite files will cause re-running of all tests, as the presented method cannot prove it safe. Another option would be to also track the execution within the test suite. That would increase the amount of storage required by the test coverage information, but the increase would most likely be minimal. Usually the test code is simpler than the tested application code, so the coverage information would only increase little.

To be able to know which application source code lines can get executed during each test, we are running the test and tracing its execution path. This path will cover all the application source code lines that can get executed during that test assuming that all the executed source code was deterministic. If there is any source of nondeterministic behaviour in the executed source code, the test coverage may not cover all possibly executed application source code lines. We will assume that all the executed source code is deterministic. Dependency libraries may exhibit nondeterministic behaviour internally as long as the nondeterministic behaviour does not change which application source code lines get executed.

The source code execution tracing is done by Python library called Coverage.py[3]. The library internally uses function settrace[4] from the sys module of the standard library. The settrace function allows setting a trace function that gets invoked before any source code lines are executed. The coverage tracing library will handle tracking and storing of the executed lines as the test suite is being executed. The library will determine which of the executed source code lines belongs to the application and which to some other library. The execution trace inside dependency libraries is ignored as explained earlier in this section. The coverage tracing library will output us list of source code files that were executed and also exact executed line numbers for each of those source code files.

The situation with source code inside templates is not so simple. As the template code is not pure python code, the Python interpreter will never directly run that code. The template code is executed by a library called

[3]https://pypi.python.org/pypi/coverage

[4]https://docs.python.org/3/library/sys.html#sys.settrace

Jinja2[5]. That library is responsible for execution of all template logic, and at this point we know of no reliable ways of making the coverage tracing library report the coverage in templates. There have been attempts to implement special plugin to the tracing library that would make the tracing library understand which template lines gets executed, but so far we know of no reliable and robust method of achieving this. The fact that our implementation can not track the coverage inside the templates is not a major issue. All the template files that are executed need to be read from the filesystem before they can be executed. These file access operations will be tracked and marked as dependent files as explained in Section 3.3. This will reduce the implementations granularity from functions to files, but still allows us to do safe test selection. Special case may be needed to make sure that the application code will not cache any template reads between tests. Any such caching would be against the assumption made in Section 3.1, and might lead to unsafe test selection.

3.5 Transforming execution path to scopes

In analysis phase we need to be able to map each source code line listed in the execution trace mapping to code scope. Different granularities can be chosen for the scope. Some examples for scope granularity are statement, function and module level granularity. For this implementation we have chosen to use function level granularity. It offers better precision during test selection phase but causes less overhead to selection phase compared to statement level granularity. Example of our a mapping is shown in Listing 3.3. The comments after each source code line represent the scope that the line would be assigned to. The code scope algorithm simply selects all source code lines that form a function body and assigns them to a scope that is named after the function name. For example in the listing source code lines 4-5 form the function body for the function `foo`. The name binding on line 3 is not part of the function body. When the function is inside class, the name of the class followed by a single dot character is added to the scope name as a prefix. This can be seen in function `qux` on line 18. The source code statements inside a function body can never form a new scope. For example the function body of `baz` on line 13-14 can not form its own scope, as it is already inside the function body of function `bar`. Therefore the function `baz` is completely inside the the function body for function `bar`.

All source code lines that do not belong to any function's body get as-

[5]https://pypi.python.org/pypi/Jinja2

signed to scope named `global`. Changes to `global` scope force the test selection algorithm to select all tests, as all tests execute some code that is assigned to `global` scope.

Listing 3.3: Example of function scopes for source code lines

```
1  a = 13                          # global
2                                  # global
3  def foo():                      # global
4      print('hello')              # foo
5      return 1                    # foo
6                                  # global
7  class B(object):                # global
8      b = 13                      # global
9                                  # global
10     @staticmethod               # global
11     def bar():                  # global
12         def baz():              # B.bar
13             value = 14          # B.bar
14             return value        # B.bar
15         return bas()            # B.bar
16                                 # global
17     def qux(self):              # global
18         return bas()            # B.qux
```

As the scope's name is directly formed from the function's name and the possible class name, there is a possibility for a scope name collision when same function name is used in more than one source code file. In these cases the scopes are considered equal, and the test selection phase is more likely to select some unneeded tests cases. This is not considered to be a problem, as it is assumed that such cases are rare.

3.6 Mapping change set to code scopes

During the test selection phase we need to be able to calculate the code scopes for each changed code line in the change set. The change set contains information about additions and removals of source code lines. In the simplest case the change set only adds or removes source code lines from within a single function's body. In more complex cases whole functions can be added or removed from multiple files. The algorithm for finding scopes with changes consists of the following steps:

1. For each source code file that has been modified compared to the previous version, take the previous version of the file and extract the list of source code line numbers where the source code line has been removed.

2. For each of those files, take the previous version of the file and calculate the code scopes as explained in Section 3.5.

3. Create a set of scopes by selecting the scopes from step 2 where the line number matches the one from step 1.

4. For each source code file that has been modified compared to the previous version, take the current version of the file and extract the list of source code line numbers where the source code line has been added.

5. For each of those files, calculate the code scopes as explained in Section 3.5.

6. Create a set of scopes by taking the scopes from step 5 where the line number matches the one from step 4.

7. Create a set of scopes by selecting the union of sets from steps 3 and 6. This set contains the scopes that have been modified when compared to the previous version.

For steps 1 and 4 we will get the change set from Git versioning system. The previous version mentioned in the algorithm steps references to the previous version of which we have the coverage information available. There may be multiple sequential commits in the Git repository that are not tested. Similar method of using the change set differences to determine the regression tests that need to be run is not a novel idea, as there is at least one existing solution using it [28].

Changes to test suite can be handled by simply rerunning all the tests that contain modifications. This works with the assumption that test files will not import code from other test files. If that assumption does not hold, we would need to also track the test coverage inside the module containing the tests in the test suite. Tracking the coverage inside test module would require small changes to the presented application. This feature was not needed for the tested application, so it was omitted.

Any change to any non-code file can be mapped to tests by using the file dependency information that is recorded during test execution. If the file with changes is not used by any tests, we have two different options how to proceed. One option is to assume that the file content is irrelevant to the test suite and the change will not require rerunning of any tests.

For example usually all written documentation within the code repository does not affect any test results, and with these changes re-running of any tests is not necessary. The other option is to assume that the file could change the result of any test, so we have to re-run all tests. The choice which option to take would require us to somehow determine if the file is important or not. Example of such important a file is `requirements.txt` file. When a program called `Pip` is used as a Python package manager to install application's dependency libraries, `requirements.txt` file will be used to define all library dependencies that the application has. Changes to that file can affect the installed libraries and so may invalidate any test results where we used libraries. As we don't track any coverage information within libraries, we must assume that all tests can be affected and thus re-run all is the only option. The presented implementation always follows the pessimistic approach where it re-runs all tests after change to file without dependency information.

3.7 Limitations

To be able to know which code sections will be run by a test, the execution trace while running that test must be deterministic, meaning that executing the test should always execute the same code lines in the same order and return the same result. Otherwise the recorded scope lists might not contain some visited code sections, and changes to those section might not trigger rerunning of the test.

During any integration test the whole application source code base gets imported. If there are any changes to any definitions (modifications to classes or global scope), some other part might use reflection or some other way to change the code behaviour. Therefore changes to global scope or class members will always require full test set execution. Example of such case is presented in Listing 3.4. If we uncomment the line 2, the output of test `test_foo` changes. Similarly by uncommenting the lines 21-23 we introduce a new function to the `Processor` class, and that addition will change the output of test `test_unknown_function`.

Listing 3.4: Example of global code change affecting test execution path

```
a = 13
# a = 15

def foo(b):
    return a + b

def test_foo():
    assert foo(1) == 14

class Processor(object):
    @staticmethod
    def process(func_name, value):
        if hasattr(Processor, func_name):
            return getattr(Processor, func_name)(value)
        return value

    @staticmethod
    def add_one(value):
        return value + 1

    # @staticmethod
    # def add_two(value):
    #     return value + 2

def test_add_one():
    assert Processor.process('add_one', 12) == 13

def test_unknown_function():
    assert Processor.process('add_two', 12) == 12
```

Changes to dependency libraries lead to rerunning all tests. This is performance limitation as tracing dependency code during test execution would lead to lower performance and greatly increase the space needs to store the executed scopes for each tests.

It is assumed that all file access operations are done through `open`-function. If files in the file system are accessed in any other way, it is not tracked by our implementation. This leads to incomplete file dependency extraction which may cause invalid test selection. This may result either in incorrectly small test set selection, or unnecessary re-test all situation. The re-test all

happens if the accessed file is not found from the file dependency database, meaning that no other test execution opened that file. If the file is found from the file dependency database, meaning that that the execution of some other test uses the `open`-function to read its contents, the incorrectly small test selection may happen. The reason for this is explained in Section 3.6.

Chapter 4

Results

In this section we are going to present the test arrangements and results for our safe test selection method. We are testing the presented method's selection results and also the performance aspects of the analysis and test selection phases. The questions this section will answer to are:

- In how many percent of test suite runs we managed to deselect some tests?
- When tests deselection worked, how many percentage of tests did it manage to deselect?
- How much overhead does the analysis phase takes compared to full test suite runtime?
- How much time does the test selection process takes on average?

4.1 Tested application

The implementation was tested against proprietary web application that has been under active development for 6 years. The main purpose of the application is to store and process user fed information, and generate different kinds of reports out of that information. The application is written in Python and consist of about 100,000 lines of source code handling over 300 different endpoints. The definition of endpoint is explained in the Section 1.1. At the time of testing the used Python interpreter was CPython 2.7.11. The application has dependency to a bit over 100 different open source code packages, of which several have been originally written for this application. About the half of the application code is located in template files. Templates consist of basic HTML structure and mix of Python code that constructs HTML

representation for the presented data. Templates are used to generate both the web pages and the reports.

The application's test suite has about 11,000 test cases and it takes about 3 hours to run it on the used development machine. The line coverage of the entire test suite is over 90%. The test suite consists of unit tests testing small portions of the application and integration tests that test the behaviour of one or more simulated requests. There are also some tests that validate the structure of some certain data files stored among the code files.

4.2 Test arrangements

The presented test selection method is tested with the application presented in the previous section. The selection algorithm performance can be measured by tracking the percentage of change sets where the algorithm can rule out some test cases. Of the cases where the method failed to deselect any test cases we manually tracked the most common reasons for the failure. The deselection failure error results were grouped separately for bug fixes and features, as to measure whether new features are more likely to make code changes that cause the coverage based tests selection to fail.

Another performance metric is to track how large of a portion of the test cases the method managed to leave unselected. Due to timing constraints we were not able to track the execution time for each selected test. We instead only measure the portion of tests that was selected and assume that the average execution time of selected tests is relatively close to the execution time of unselected tests. With this assumption we can approximate the actual time taken by test set execution from the portion of tests selected in each case.

The change sets used in the performance evaluation are extracted from the application source code repository. The tested change sets are all the differences in the repository between two consecutive application version. Each tested change consist of one or several code change commits done to address a single item in the issue tracking system. The code change sets are always labeled either as a bug fix or a new feature. With each application version the test set passes without finding any faults. The performance evaluation was done to the last 500 application versions. Of these versions 181 were done to fix a bug in the application and 319 versions contained general improvements or new features.

Third measured performance metric is how much more time it takes to run the test set with the execution tracing and how much time it takes to select the changed tests. The time taken by execution tracing depends on

how much of the test set execution time is used inside the application code. As explained in the Section 3 the execution trace inside dependency packages' source code is not tracked. This allows us to store less execution trace data and reduces the time added by execution tracing.

4.3 Test deselection results

Test deselection is a case where the test selection algorithm decided to not run the deselected tests, because the test case had no mutation on its execution path or inputs, and so could not be fault revealing. Test deselection is called successful if the algorithm managed to deselect even one test case. In unsuccessful test deselections the algorithm could not deselect any tests, meaning that all tests in the test suite had to be re-run.

The results for issues where no test cases could be deselected are listed in the Table 4.1. The average test's deselection success rate was 34%. Of the deselection failures 52% were caused by code structure modifications. Faced with structure modifications the coverage based method simply cannot make any safe deselections, as the modification done in the change set can have an effect already when the application code is being imported. The second largest deselection error source with 4% of the cases were modifications to code dependencies. As explained in the Section 3.7 the execution path in the code dependency libraries is not being tracked, so the information about which tests access which libraries is not available. Therefore any library change forces the safe test selector to fall back to use re-run all method. As the dependencies change so rarely, storing the execution trace information for them is not important.

When the change sets done for bug fixes are separated from the ones done for features and improvements, the results change significantly. The bug fix changes are less likely to make code structure modifications, and this leads to to test deselection success rate of 47% for those change sets. For the feature change sets the deselection success rate dropped to 26%, which is mostly caused by the increased 59% portion of code structure modifications. The portion for deselection failures caused by change to code dependencies didn't change significantly.

For the succeeding test case deselection cases, the number of selected tests is shown in Figure 4.1. In the majority of these cases the deselection method managed to deselect vast portion of the tests. In most cases there were very few test that needed to be run. This is reasonable as a change to a single endpoint's code does not require running any tests for any other endpoint.

The number of tests visiting each code scope is visualised in Figure 4.2.

	#For all	**#Bug fix**	**#Feature**
Total test suite runs	500 (100%)	181 (100%)	319 (100%)
Some test cases can be unselected	169 (34%)	85 (47%)	84 (26%)
Re-run all: Code structure modification	261 (52%)	74 (41%)	187 (59%)
Re-run all: Modification to dependency	20 (4%)	6 (3%)	14 (4%)

Table 4.1: List containing coverage based test selection test deselection performance and most common reasons for its failure.

As the scope granularity was chosen to function level granularity, the graph shows how many tests actually execute each function in the application. The lowest recorded number of tests covering a specific function was zero, meaning that some functions were not covered by any test. The highest number of tests executing single function was over 11,000 meaning that each test in the test suite executed that function. When the scopes are ordered by the number of functions visiting them, it can be seen that the grow rate in number of visiting tests is behaving logarithmically.

From the same Figure 4.2 we can see that the percentage of tests that cover arbitrary scope is on average very low. Over 70% of scopes are being visited by 1% or less of the test cases. Also just 5% of all scopes that are visited by more than half of test cases. Those scopes are mostly executed during the test initialisation, and are responsible for starting up the application for testing. As seen from the Figure 4.1 change sets containing modification to those sections are relatively rare, as the number of test suite runs where large portion of tests were selection was low.

4.4 Test performance results

The presented method consist of two phases: Analysis and tests selection. The analysis phase runs all the tests while tracing each test's coverage, so it imposes some performance overhead compared to simply running all tests. When running on a development machine the test suite takes 2h 7min and with the tracing the time increases to 4h 20min. So the analysis phase roughly doubles the execution time compared to a normal rerun all case.

The test selection process performance depends on how large the change set is and how many rows there are stored in the coverage database. If the analysis phase have been done to multiple application versions, the coverage rows for each versions are stored in the same database. With single applica-

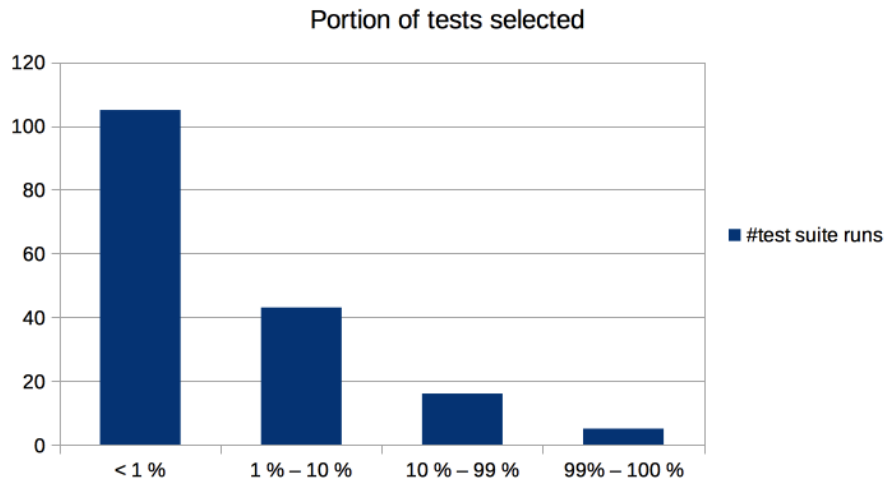

Figure 4.1: Approximation of portion of tests selected in test suite runs.

tion versions coverage information in the database the test selection always took less than a few seconds, so the overhead is not significant. Selection performance was not tested when the coverage database had coverage information from multiple different application versions. If the performance of the coverage database would become a problem with multiple application versions, it is easy to split the database so that each application version stores its coverage information to separate database file.

The coverage database with coverage information of single application version used in total 345 MB of space. The database contained in total bit over 1.7M rows. Of these rows around 1.6M were used to store link between scopes and tests and 0.1M to store file access operations in tests.

4.5 Discussion

The presented coverage based test selection method proved to select only small portion tests when the change set didn't make changes to the application structure. The most significant issue with it is that with the tested application over 50% of the change sets had application code structure changes as can be seen from the Table 4.1. Code structure changes can make major differences to the tests' coverage, and so prevent the presented test selection method from deselecting any tests.

Smaller improvements to the granularity of template files could also be done. Template files contain mix of Python code and HTML code. The test

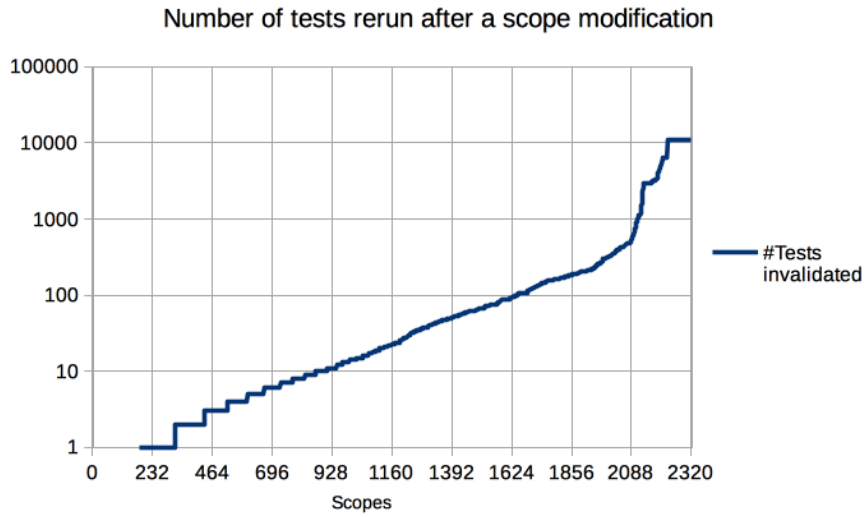

Figure 4.2: Number of tests using scopes.

execution trace method used can not track the coverage inside the template files. There is a plugin that would allow tracking on template files, but its coverage results proved to be too unreliable to use in this context. From the file dependency information we get low granularity information on which template file changes would require selection of which tests. Dependency files can not understand the changes inside templates, and so any change to a template would select all test cases where that file is accessed. With template coverages we could achieve the same granularity as in python code, so changes to any function would only select tests that visit that function.

When the test selection selects small portion of the tests it is easy to keep the test scopes up to date. When faced with re-run all situation, creating the test scopes would require the execution of all test cases, and with the tested application that would take several hours on developers' machines. With the measured code structure modification rate the presented code selection method cannot make it feasible to run the tested application's test suite on developers' machines.

If the analysis phase was being calculated on the Continuous Integration servers, the developers machines could use that database when doing the test selection. This would remove the constant re-run all tests case when new change set with code structure modification gets merged to the used source code base.

In the cases where developer's change set would make code structure mod-

ifications, the test selection method could ignore the code structure changes and use the other scopes with changes to do the selection. This turns the whole selection algorithm to unsafe to make it more performant. This way the developer is not facing the re-run all test cases situation, but rather left with some tests that could detect a fault in the change set. The full re-run all process could be handled in the CI server side.

Modification to dependency libraries is a change where we do not have test scopes available. As seen from the results the dependency libraries change relatively rarely, that re running all test cases in this case is completely feasible solution. Tracking and storing test scopes in dependency code would slow down the test suite execution and increase the storage space requirements for test scope information.

Chapter 5

Conclusions

In this paper we tried to find a solution to the performance issues in regression testing when facing ever increasing amount of test cases. We researched existing work related to test selection methods, and came to the conclusion the the current test selection algorithms are not suitable for dynamic programming languages, such as Python. Our proposition consists of an algorithm that tries to select a small subset of test is the test suite in a way that the selected tests still find all the possible faults that the recent modification caused.

The presented algorithm works by tracing the test suite execution and creating a map about which tests execute which functions. With the assumption that the tested application is deterministic it can be seen that a test does not need to be re-executed if the change set only contains modifications to the contents of unrelated functions. Unfortunately this algorithm cannot deselect any tests if the change set has any modifications to functions definition or the general structure of the source code. Similarly the algorithm also keeps track of files opened by tests, and so can choose fault-revealing test subset in cases where the results of some tests may depend on the contents of some files. This feature is important in web applications where template files are often used to construct responses to requests.

The proposed algorithm can be integrated into existing applications test suite without any modifications to the source code of the tested application. In our work we tested our test selection algorithm against the historical versions of existing proprietary medium sized web application, and got some conflicting results. The test selection algorithm managed to deselect large portions of the test cases when the change sets made changes within functions. Unfortunately roughly the half of the change sets contained changes to function definitions and other structure of the source code, making the test selection algorithm fail to deselect any test cases. Also the implementation

had major negative effect on the test suite runtime.

5.1 Future work

As discussed in Section 3.6 there needs to be some logic that decides which change sets require re-running test. The presented implementation chose the pessimistic approach where a change to file without dependency information required us to re-run all tests, because there exists files where that is the only option. We could replace that pessimistic approach with some heuristics that would somehow detect if the changed files are important, and so reduce the number of re-run all occurrences. It could also be researched whether cleaning the change set would help us reduce the number of changes sets that make changes to global scope. Cleaning in this context means ignoring all non-code aspects of the code, such as changes to code comments or any changes that have no effect to the code's execution. Such cleaning has been done in some previous research [28] where the code was transformed to canonical form before the change set was calculated, but it is unclear how much effect it has to test selection performance.

In existing research it has been noticed that the test selection performance is sensitive to the program and its test suite [11]. The characteristics of web applications might be better or worse for coverage based test selection. Therefore one research case would be to implement similar test selection methods to existing non-web applications and report the actual selection performance results. So far most research has used example applications that are small and do not represent the characteristics of real applications that are being used in the field. Similarly it could be researched how the tracing granularity affects the test selection performance. In the presented algorithm we chose to use function level granularity without any proof that it would be the best option. The best granularity might also be affected by the characteristics of the tested application. With function level granularity the test tracing performance could be improved by using setprofile[1] interpreter hook that is called less often the one used by the implementation. The other performance overhead could most likely be mitigated by normal optimization tricks.

This experimental test selection performance done in Section 4 used the historical versions of a proprietary application, but this test does not show that the test selection algorithm always chooses the fault revealing test cases. Because of the development processes used during the application develop-

[1] https://docs.python.org/3/library/sys.html#sys.setprofile

ment, the test suite of every single tested version of the application passes without finding any faults. Therefore we can not take this test as an indicator that the test selection method would find all failing test cases. In general case there might be some aspect to the testing that makes the presented method unsafe. Even though proving that the test selection is safe might not be feasible, testing the algorithm in real life situation would increase confidence in the presented algorithm.

As seen from Chapter 4.3 the tested application's history contained lots of changes that changed the structure of source code. In some cases it might be preferred to ignore the made changes to the code structure, and just run the test cases that execute functions that are changed in the change set. This makes the algorithm unsafe, meaning that it might not select all fault revealing test cases. The positive effect of this change is that it would make the algorithm to choose small subsets more often. In some cases it might be preferred that the developer can locally execute the small, relevant test subset and let the CI handle running the full test suite.

Bibliography

[1] Agrawal, H., Horgan, J. R., Krauser, E. W., and London, S. Incremental regression testing. In *Proceedings of the Conference on Software Maintenance, ICSM 1993, Montréal, Quebec, Canada, September 1993* (1993), D. N. Card, Ed., IEEE Computer Society, pp. 348–357.

[2] Beazley, D. M. *Python essential reference.* Addison-Wesley Professional, 2009.

[3] Ben-Asher, Y., and Rotem, N. The effect of unrolling and inlining for Python bytecode optimizations. In *Proceedings of of SYSTOR 2009: The Israeli Experimental Systems Conference 2009, Haifa, Israel, May 4-6, 2009* (2009), M. Allalouf, M. Factor, and D. G. Feitelson, Eds., ACM International Conference Proceeding Series, ACM, p. 14.

[4] Biswas, S., Mall, R., Satpathy, M., and Sukumaran, S. Regression test selection techniques: A survey. *Informatica (Slovenia) 35*, 3 (2011), 289–321.

[5] Chen, Y., Rosenblum, D. S., and Vo, K. Testtube: A system for selective regression testing. In *Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy, May 16-21, 1994.* (1994), B. Fadini, L. J. Osterweil, and A. van Lamsweerde, Eds., IEEE Computer Society / ACM Press, pp. 211–220.

[6] Chen, Z., Chen, L., Zhou, Y., Xu, Z., Chu, W. C., and Xu, B. Dynamic slicing of python programs. In *IEEE 38th Annual Computer Software and Applications Conference, COMPSAC 2014, Vasteras, Sweden, July 21-25, 2014* (2014), IEEE Computer Society, pp. 219–228.

[7] Elbaum, S. G., Malishevsky, A. G., and Rothermel, G. Prioritizing test cases for regression testing. In *Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2000, Portland, OR, USA, August 21-24, 2000* (2000), D. J. Richardson and M. J. Harold, Eds., ACM, pp. 102–112.

[8] Elbaum, S. G., Malishevsky, A. G., and Rothermel, G. Test case prioritization: A family of empirical studies. *IEEE Trans. Software Eng. 28*, 2 (2002), 159–182.

[9] Engström, E., Runeson, P., and Skoglund, M. A systematic review on regression test selection techniques. *Information & Software Technology 52*, 1 (2010), 14–30.

[10] Fritz, L. Balancing cost and precision of approximate type inference in Python.

[11] Graves, T. L., Harrold, M. J., Kim, J.-M., Porter, A., and Rothermel, G. An empirical study of regression test selection techniques. *ACM Transactions on Software Engineering and Methodology (TOSEM) 10*, 2 (2001), 184–208.

[12] Gupta, R., Harrold, M. J., and Soffa, M. L. An approach to regression testing using slicing. In *Software Maintenance, 1992. Proceerdings., Conference on* (1992), IEEE, pp. 299–308.

[13] Hwu, W. W., and Chang, P. P. Inline function expansion for compiling C programs. In *Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation (PLDI), Portland, Oregon, USA, June 21-23, 1989* (1989), R. L. Wexelblat, Ed., ACM, pp. 246–257.

[14] Jones, J. A., and Harrold, M. J. Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE Trans. Software Eng. 29*, 3 (2003), 195–209.

[15] Kim, J., and Porter, A. A. 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, 19-25 May 2002, Orlando, Florida, USA* (2002), W. Tracz, M. Young, and J. Magee, Eds., ACM, pp. 119–129.

[16] Leung, H. K., and White, L. A study of integration testing and software regression at the integration level. In *Software Maintenance, 1990, Proceedings., Conference on* (1990), IEEE, pp. 290–301.

[17] Li, Z., Harman, M., and Hierons, R. M. Search algorithms for regression test case prioritization. *IEEE Trans. Software Eng. 33*, 4 (2007), 225–237.

[18] Loelinger, J., and MacCullogh, M. *Version Control with Git - Powerful Tools and Techniques for Collaborative Software Development: Covers GitHub, Second Edition.* O'Reilly, 2012.

[19] Myers, G. J., Sandler, C., and Badgett, T. *The art of software testing.* John Wiley & Sons, 2011.

[20] Namiot, D., and Sneps-Sneppe, M. On micro-services architecture. *International Journal of Open Information Technologies 2*, 9 (2014), 24–27.

[21] Pan, J., and Center, L. T. Procedures for reducing the size of coverage-based test sets. In *Proceedings of International Conference on Testing Computer Software* (1995), Citeseer.

[22] Rothermel, G., and Harrold, M. J. Analyzing regression test selection techniques. *IEEE Trans. Software Eng. 22*, 8 (1996), 529–551.

[23] Rothermel, G., and Harrold, M. J. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Methodol. 6*, 2 (1997), 173–210.

[24] Rothermel, G., Harrold, M. J., and Dedhia, J. Regression test selection for C++ software. *Softw. Test., Verif. Reliab. 10*, 2 (2000), 77–109.

[25] Rothermel, G., Harrold, M. J., Ostrin, J., and Hong, C. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *1998 International Conference on Software Maintenance, ICSM 1998, Bethesda, Maryland, USA, November 16-19, 1998* (1998), IEEE Computer Society, pp. 34–43.

[26] Rothermel, G., Untch, R. H., Chu, C., and Harrold, M. J. Prioritizing test cases for regression testing. *IEEE Trans. Software Eng. 27*, 10 (2001), 929–948.

[27] Srivastava, A., and Thiagarajan, J. Effectively prioritizing tests in development environment. In *Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2002, Roma, Italy, July 22-24, 2002* (2002), P. G. Frankl, Ed., ACM, pp. 97–106.

[28] Vokolos, F. I., and Frankl, P. G. Pythia: A regression test selection tool based on textual differencing. In *Reliability, quality and safety of software-intensive systems.* Springer, 1997, pp. 3–21.

[29] White, L. J., and Leung, H. K. A firewall concept for both control-flow and data-flow in regression integration testing. In *Software Maintenance, 1992. Proceerdings., Conference on* (1992), IEEE, pp. 262–271.

[30] Yoo, S., and Harman, M. Regression testing minimization, selection and prioritization: a survey. *Softw. Test., Verif. Reliab. 22*, 2 (2012), 67–120.