Limiting Transitive Closure for Static Regression Test Selection approaches

HENRIK KARLSSON
Limiting Transitive Closure for Static Regression Test Selection approaches

HENRIK KARLSSON
Abstract

In computer science and software development it is important to test software in order to ensure reliability. Regression testing in order to find potential faults introduced by software changes is key to assuring that the software is stable. This process may be time consuming, so in order to speed it up there are approaches which select a subset of relevant tests based on the software changes.

In the field of regression test selection there are two main approaches, static approaches and dynamic approaches. These different kinds of approaches have different strengths and weaknesses, however the field is currently dominated by dynamic approaches as the performance of the static approaches lags behind severely. Ensuring that the correct approach types are used for the appropriate situations calls for improvement of static approaches.

Regression test selection approaches uses transitive closure to select relevant tests. For any node in a directed graph, transitive closure is the set of all reachable nodes from the starting node. This thesis proposes a solution which attempts to lessen the performance gap by implementing a controlled limit to the transitive closure property of the main test selection algorithm. The aim of the limited transitive closure is to reduce the time taken to select tests, and to reduce the amount of superfluous tests selected.

The results show that the limited transitivity property as implemented for this thesis did not improve the performance in a satisfactory way. Safety dropped severely. Since the runtime and precision improved, there is room for improvement in potential future research of the limited transitivity approach before it can readily be dismissed as an unviable approach.
Sammanfattning

Inom fälten datalogi och mjukvaruutveckling är det viktigt att testa mjukvara för att försäkras sig om att den fungerar som den ska. Att köra regressionstester för att hitta potentiella fel som introducerats av mjukvaruändringar är en nödvändighet för att garantera stabiliteten hos mjukvara. Regressionstestprocessen kan dock vara resurs- och tidskrävande, vilket har lett till introduktionen av urval av tester som körs vid regressionstestning, s.k. regression test selection.

Fältet regression test selection delas allmänt upp i två olika typer av metoder, statiska och dynamiska. De statiska och dynamiska metoderna har olika styrkor och svagheter, och bör därför i teorin användas baserat på vilken typ som bäst lämpar sig för syftet. Dock är det stora prestandaskillnader och därför används nästan bara de dynamiska metoderna.

I detta examensarbete presenteras ett försök till att öka prestandan av de statiska metoderna genom att använda sig av en kontrollerad begränsning av transitivitetsaspekten inom huvudalgoritmen för urval av tester. Syftet med denna kontrollerade begränsning är tvådelad; både att minska körtiden för testurval, men också att minska antalet överflödiga tester i det slutgiltiga urvalet.

# Contents

1 Introduction .............................................. 1  
   1.1 Purpose and Hypothesis ............................ 2  
   1.2 Goals ............................................. 2  
   1.3 Research Questions ............................... 2  
   1.4 Thesis Layout ..................................... 3  

2 State of the Art ........................................ 4  
   2.1 Regression Testing ................................. 4  
   2.2 Class Firewall ..................................... 5  
   2.3 Safety .............................................. 6  
   2.4 Regression Test Selection ......................... 7  
   2.5 Dynamic Approaches ................................ 9  
   2.6 Static Approaches .................................. 9  
   2.7 Granularity .......................................... 9  
   2.8 Related Research .................................. 10  

3 Approach .................................................. 13  
   3.1 Novelty .............................................. 13  
   3.2 Finding Dependencies ............................... 14  
      3.2.1 Gathering Dependencies ....................... 14  
      3.2.2 Creating the Transitive Dependency Graph .... 15  
   3.3 Version Control ..................................... 17  
      3.3.1 Checksum Verification ......................... 17  
      3.3.2 Checksum Comparison ......................... 18  
   3.4 Selecting Tests .................................... 18  
      3.4.1 Identifying Dependencies ..................... 19  
      3.4.2 Running tests .................................. 20  
   3.5 Implementation .................................... 20
4 Experiment Execution and Results
  4.1 Protocol ................................................. 21
    4.1.1 Creating a Dataset ............................... 22
    4.1.2 Selecting Projects ............................... 22
    4.1.3 Filtering Revisions ............................... 23
    4.1.4 Running Experiment Software ................... 24
    4.1.5 Recording Results ............................... 24
    4.1.6 Summarizing Data ............................... 24
  4.2 Approach .................................................. 26
  4.3 Running the Experiment ................................. 27
    4.3.1 Projects Chosen ................................... 27
    4.3.2 Physical Setup for Experiment Run ............... 27
  4.4 Results .................................................... 28
    4.4.1 Metrics ............................................. 28
    4.4.2 Dataset ............................................. 28
    4.4.3 Empirical Results ................................ 29
  4.5 Problems Encountered During Experiment Run ............ 36

5 Discussion .................................................. 38
  5.1 Tests Selected ......................................... 38
  5.2 Runtime ................................................ 39
    5.2.1 Runtime Compared to AllTests .................... 39
    5.2.2 Test Selection Portion ............................ 39
  5.3 Safety .................................................. 40
  5.4 Precision ................................................ 40

6 Threats to Validity ........................................ 42
  6.1 Threats to Internal Validity ............................ 42
  6.2 Threats to External Validity ............................ 43
  6.3 Threats to Construct Validity .......................... 44

7 Conclusions .................................................. 46
Chapter 1

Introduction

In computer science and software development it is important to test software in order to make sure it is working as intended. This can take a lot of time to do manually so projects, especially those with large code bases, run test suites. These suites contain a multitude of tests in order to reach as much of the code in the project as possible, since as the number of tests increase so does the number of potential discovered bugs. Therefore it is of interest to have as little untested code as possible in a software project.

Running a standard set of tests can be very inefficient as a lot of the tests run may be superfluous when they run on code which has not been changed [1]. Regression test selection prevents this by detecting what code has been changed, and what code relates to the changed code. A relevant subset of tests is then chosen to run based on what code was changed. The subset of tests selected to be run will then only contain those tests which may detect faults caused by the code changes.

Regression test selection approaches are largely split into two categories, static approaches and dynamic approaches. The key difference between the two kinds of approaches is that the static approaches calculate the tests to be selected from the software dependencies each time they are run, while the dynamic approaches base the selected tests partially on the data from previous runs.
1.1 Purpose and Hypothesis

This research is done in order to assess whether it is possible to improve the performance of Static approaches for regression test selection such that they may compete with Dynamic approaches in terms of runtime, test selection, precision, and safety.

The question this thesis will attempt to answer is “Can static regression test selection approaches at the class-level granularity reach the safety level of dynamic approaches without negatively affecting speed and precision?”.

Hypothesis: it is possible for static regression test selection approaches to reach the safety of the dynamic ones without sacrificing execution speed or precision. This is based on the fact that the state of the art research paper [1] shows that Static approaches can at least approximate the performance of Dynamic approaches, with only slightly worse performance in runtime and safety.

1.2 Goals

The objective of the thesis is to further advance the knowledge within the domain of regression test selection, specifically the static approach solutions at the coarse (software class-level) granularity level. This will be done by introducing a new limited transitivity property and observing how it affects the performance of a static regression test selection approach, by comparing it to the performance of standard approaches of both static and dynamic types.

The two static approaches to be used are STARTS, which is the result of the state of the art research [1], and HKRTS, which is an approach that implements the new property researched in this thesis. For this experiment the Ekstazi [6] tool will be used as a representative for the dynamic approaches, because it has proven to be useful for this purpose in previous research [1] [2].

1.3 Research Questions

In order to fulfill the objective of this thesis, the research questions needed to be answered are how the limited transitivity property affect the following for a static regression test selection approach:
1. Number of tests selected
2. Runtime
3. Relative safety
4. Relative precision

In order to evaluate the research questions data will be gathered from open source software projects. These projects will be taken from GitHub.

1.4 Thesis Layout

The second chapter contains a background description on the area of regression test selection. It goes into detail of the different kinds of approaches, and explains their differences.

The third chapter contains a detailed description of how the HKRTS software was implemented.

The fourth chapter contains the goals for the experiment, the experimental approach, and the results of the experiment.

The fifth chapter contains discussion of the effects and implications of HKRTS may have based on the experimental results.

The sixth chapter contains discussion regarding potential threats to validity.

The seventh chapter contains the conclusions drawn from the results, and what possible future research could entail.
Chapter 2

State of the Art

2.1 Regression Testing

The process of testing for effects of code changes using existing tests is called regression testing [12]. When code is changed it needs to be tested again in order to make sure that it is not failing, and is up to the standards set for the project it is in. This is done by having a test suite which may be run on the code in the project after changes have been made. The purpose of regression testing is to discover the potential erroneous behaviour which may be introduced into a program or piece of code which previously ran without errors.

A large-scale study was conducted by Beller et al. [5] with the purpose of mapping developers’ testing behavior to tangible data. Beller et al. convinced developers to use the Eclipse IDE plugin WatchDog, which allowed them to gather activity data from the developers IDEs. The data shows that developers spend close to 75% of their time on production, and 25% of the time on testing. However, the developers estimated an almost equal division between production and testing. The results of the research were statistically significant, which implies that overall developers only test about half as much as they estimate.

Gligoric et al. [4] conducted research with the aim to study the extent to which manual regression testing was performed, and how well manual regression testing performed compared to automatic regression testing. What they found was that manual regression testing chose to run a superfluous amount of tests 73% of the time. In addition, their results showed that in 74% of the test sessions manual regression testing did not select all tests affected by recent code changes.
According to the paper by Biswas et al. [28], approximately half of all software maintenance costs comes from regression testing. An example of this resource cost is the study by Do et al. [29], wherein they report that it took 1,000 machine hours to run the tests for a software product with 30,000 test cases. This presents a need not only to perform regression testing, but also to make sure that the tools used are cost effective in order to minimize the resources needed for regression testing.

2.2 Class Firewall

Back in 1990 Leung et al. [18] researched the concept of integration testing and in their paper laid the ground works for the area of regression test selection. In their attempt to increase the efficiency of integration testing they proposed the concept of a “Firewall” to encapsulate the program modules which may potentially be affected by code changes. The use of this firewall is to limit the amount of tests required to run in the integration testing process after code changes has been made, by selecting a subset of tests based on what code was changed. Comparing their test selection approach to running all tests showed that test selection ran only 35% of tests.

In the wakes of the research of Leung et al., Rothermel et al. [16] conducted research in order to find an algorithm which could be used in a regression test selection technique to ensure that all tests which possibly could detect regression faults are selected. They found that their algorithm was not only safer than previously proposed techniques, but also more precise. This was thought to stem from the fact that their technique selected tests based on the changed code with the help of a graph of dependencies, rather than looking for relevant tests in a predetermined test suite.

In a subsequent research study Rothermel et al. [15] implemented their theoretical regression test selection algorithm [16] into a tool. They found that the implementation of their algorithm was as safe as they had previously theorized. In addition to this they also found that the size of the software matters when applying regression testing; the regression test selection technique was more efficient at saving time when working on large and complex programs than when working on smaller simple programs. They also found that there are some pro-
grams (with corresponding test suites) where regression test selection will not offer savings.

As object oriented programming became more common, integration testing had to be adapted to such languages, as in the paper by Hsia et al. [30]. They proposed a process which utilizes the object oriented class firewall technique in combination with marking visited test cases, in order to create a functioning regression testing technique for the object oriented languages. Their research showed that the technique which they proposed saved up to 67.6% of the total test run time.

Skoglund et al. [11] expand upon the concept of class firewalls in a modern context. In their paper they talk about the steps required to use a class firewall selection technique in practice. They found that a class diagram of the dependencies in the software had to be created. Using this diagram, all affected classes have to be found and tested. A class is considered affected if it has either a direct or indirect dependency relation to another changed class.

In the 2001 paper by Grove et al. [17], an algorithm is presented that encompasses a large amount of call graph construction algorithms. They found that using call graphs provided speedups for a significant portion of projects tested. The usage of a call graph covers the needs which regression test selection has for a class diagram, which is to set up a traversable structure of the dependencies in the software.

Badri et al. [14] did research on how to apply call graphs to integration testing. In their paper they present a static technique which may be used to predict the impact of code changes, based on control call graphs. They found that their technique was more precise when compared to techniques without the call graph functionality. In addition they recommend the usage of call graphs together with regression test selection.

2.3 Safety

According to the paper by Öqvist et al. [24], a safe regression test selection approach is one which will select to run all tests which have been affected by code changes. This means that a completely safe regression test selection technique will result in equivalent safety as running all tests. In addition to this, a completely precise approach will select only tests which have been affected by code changes. They also state
that a regression test selection technique is more precise the fewer tests outside of the safe test set it chooses to run. Thus, a perfectly safe and precise regression test selection technique will choose to run the subset of tests which contains exactly all tests which have been affected by code changes and no other tests.

Research by Skoglund et al. [31] on the topic of change-based regression test selection presented a new approach based on the class firewall approach. They showed that their approach selected fewer tests than the class firewall approach, without suffering any safety losses. Additionally, their research proves regression test selection approaches at class level dependency analysis to be safe.

In a paper by Harrold et al. [20] they present a safe regression test selection technique that handles Java features. They describe the goal of regression test selection techniques as attempts to reduce the cost of regression testing by selecting a subset of tests and using it to test the new version of a changed software. In their research they combine the set of differences between software versions with a test coverage matrix. Their technique selects tests based on the first affected class found on each path in the difference set. The results of their study was that their technique was able to reduce the size of the test suite, as well as that regression test selection at finer granularity could prove to further reduce the test suite size.

In 2004 Orso et al. [19] conducted a study on the topic of safe regression test selection techniques. They evaluate a tool which combines the class diagram with a test selection technique which operates at statement level to find differences in code. They found that their tool reduced the time it took to perform regression testing by up to 62.5%. They also provide the statement that a cost-effective regression test selection technique has a lower cost of performing the selection and rerunning the selected subset of tests, than the cost of rerunning the complete test suite. This is a logical claim, since the aim of regression test selection is to reduce the system cost of running integration testing.

### 2.4 Regression Test Selection

In a study conducted by Legunsen et al. [1] two static regression test selection approaches were compared, one at coarse granularity
and one at fine (software method-level) granularity. These two approaches were also compared to a dynamic regression test selection tool. What was found was that static regression test selection in general was underperforming compared to dynamic regression test selection approaches. In addition at fine granularity the static approach was too slow to be viable, and at coarse granularity the static approach was comparable to the dynamic in terms of speed, but performed worse in terms of safety. The conclusion of the research was that static regression test selection needed further research, with focus on coarse granularity techniques. This thesis will conduct some further research by attempting to increase the speed and safety of static regression test selection approaches to make it a viable alternative.

Automatically selecting and running tests using regression test selection has multiple benefits. One important benefit is that of time; the developer can spend time on something else, and a computer calculates faster than a person. Another important benefit is that of consistency; an unbiased program will always select tests based on the same criteria for each run. As was found in previous studies [4], humans tend to select both unsafely and imprecisely.

Gligoric et al. [6] conducted research with the aim of attempting to make regression test selection usage more widespread. They found that out of 666 projects studied not one of them utilized automatic regression test selection techniques. They asked themselves how researchers should design a regression test selection technique in order to make it more likely to be practically used. The solution they came to was introducing a new tool called Ekstazi, a lightweight and easy to implement tool which operates dynamically with file dependencies. In contrast to the traditional regression test selection approaches, Ekstazi computes based on test approaches and test classes; if a class that a test depends on has changed, the test is flagged for running. These changes were discovered by analyzing the checksums of the classes before and after the changes. Ekstazi also utilizes coarser dependencies compared to the traditional techniques.

They state that to properly evaluate the added benefit of a regression test selection technique, the time measured should be from the execution of the regression test selection software until the developer is once again in control. With this in mind, the Ekstazi tool is on average 32% faster than running all tests, which increased to 54% for the test suites with longer run times. Despite selecting more tests than
other regression test selection techniques, such as the one by Orso et al. [19], Ekstazi had a lower end-to-end runtime.

2.5 Dynamic Approaches

According to Gligoric et al. [6], dynamic approaches of regression test selection need two things in order to operate. For any two revisions of a project r[i] and r[i+n], where i is a revision number and n the difference in revision numbers between the two revisions, the information needed by a dynamic approach is;

1. What code has changed between r[i] and r[i+n].

2. Test dependencies, which are dynamically computed during the test run on r[i].

Previous research [1] shows that dynamic approaches have the upside of having a lower runtime compared to static approaches. There is also research [6] which points to dynamic approaches being safer, and more precise, than static approaches.

2.6 Static Approaches

In contrast to the dynamic approach, static approaches only need the information of what code has changed between revisions, according to research such as the study by Legunsen et al. [2]. Test dependencies are calculated before the test run and are then used statically to find out what tests to run. Since static approaches do not require previous test dependencies like the dynamic approaches do, they can be run in some situations where dynamic approaches cannot.

2.7 Granularity

Previous research finds that the level of detail of the dependency search matters for static regression test selection approaches [1]. Searching for dependencies at coarse granularity decreases the runtime by at least 50% compared to fine granularity, with no decrease in safety or precision.
2.8 Related Research

Test case prioritization is a technique which aims to change the order of which test cases are executed, with the purpose of increasing the amount of faults found in the regression testing stage, as can be seen in such papers as Hao et al. [23]. Test case prioritization techniques either fall into the category of total strategies, or additional strategies. Total strategies prioritize tests with the highest amount of code coverage, while additional strategies prioritize tests which will cover the most of the currently uncovered code.

By combining regression test selection with test case prioritization techniques, Elbaum et al. [7] presents algorithms with the purpose of making integration testing more cost effective. What their approach entails is that they split the testing into two phases; the first phase is a run of regression test selection on a selected set of modules. In the second phase, where dependent and changed modules are tested, they employ test case prioritization on the result generated by the first phase. The end result of this combination of techniques lowers the testing load which yields an improvement in terms of resource saving.

Ryder et al. [10] propose algorithms which may help detect the impact on performance that code changes may have for object oriented software. They claim that these algorithms can be used to detect the tests which are affected by changes. They also say that in case of a test failure the algorithm can determine a set of changes which may be introduced without affecting other tests. In order for their algorithms to operate they require a control flow graph of the software classes. Since their algorithms result in a subset of classes which are affected by hypothetical changes, their algorithms are useful for regression test selection.

Closely related to the area of change impact analysis is the paper by Zhang et al. [25] regarding fault localization. In their paper they present a novel technique for ranking the effect of program edits with the purpose of reducing the manual effort of developers when inspecting code change effects. Their technique utilizes spectrum-based fault localization in combination with call graphs in order to detect what edits caused found faults. They found that their technique was slightly more efficient than the previous state of the art, while increasing the rate of fault detection. The process of fault localization is useful to combine with regression test selection in order to find not only the
Shi et al. [13] study the capabilities of Test Suite Reduction. Test Suite Reduction is a technique which aims to speed up the testing process. This is achieved by removing tests which are deemed redundant according to a set of predefined criteria, resulting in a subset of tests with a total runtime lower than running all tests. In Test Suite Reduction, a mutant is a piece of code which has changed behavior from a previous version. In an attempt to increase the efficiency of Test Suite Reduction, they propose a technique which bases the criteria on the amount of detected mutants by the selections made. This approach resulted in a stable test suite which detects the same mutants as the original test suite.

Zhang et al. [26] conducted a study with the purpose of investigating the performance of Test Suite Reduction when used on Java programs with JUnit. This was in contrast to previous research which was mainly done in C. What they found was that the Test Suite Reduction techniques reduce the test suite size of Java software without substantially lowering the safety of the test suite.

Zhong et al. [27] claims that previous experimental results on Test Suite Reduction have been limited by running on too small software programs, and not comparing techniques to each other. They conducted a study of their own in order to counteract this where they implemented a total of four common Test Suite Reduction techniques and then applied them to projects of varying sizes in order to make a comparison of the techniques. They found that all four of the general techniques were capable of dramatically reducing the size of test suites. This solidifies the idea of Test Suite Reduction as an effective integration testing tool.

Shi et al. [8] furthers the research of regression testing techniques by continuing their research of Test Suite Reduction. In this study they compare regression test selection to Test Suite Reduction in terms of ability to speed up the regression testing process. Their findings were that regression test selection selected fewer tests, and had no loss in its ability to detect possible faults introduced by code changes while there was loss present for Test Suite Reduction. They also recommend to combine regression test selection and Test Suite Reduction if there are needs to further speed up the testing approach after applying regression test selection. This combination reduced the run time more than just using regression test selection, however at the cost of intro-
ducing some loss of safety.

Panda et al. [3] uses test case prioritization in order to achieve somewhat similar effects of regression test selection. In their paper they present a static approach to test case prioritization by computing the coupling of the affected parts of changed software. They reported that the results of the experiments indicate that their technique performs at an adequate level compared to other existing techniques.

Other research, such as the one by Legunsen et al. [22], have made solutions which combines regression test selection with other techniques. Their solution uses monitoring-oriented programming, which is a framework with the purpose of lessening the gap between specifications and the actual software produced. The solution proposed has the purpose of extending the functionality to multiple software versions. The technique they propose combines the functionality of monitoring oriented programming with attributes of regression test selection. The purpose is to make it “evolution-aware” and to increase the efficiency of monitoring oriented programming by making the monitored code only be code which changed between versions. The results of their experiments with their solution shows that while their technique improved the performance, they recognised that there was still work to be done due to the size of the runtime overhead.

Bell et al. [21] conducted research on the area of dependency detection with the purpose of attempting to make the process of finding test dependencies more efficient. This was due to the fact that other regression testing techniques, such as test case prioritization and regression test selection, all assumed that tests were independent of each other during their selection processes. The authors proposed a tool which was based on the solution of monitoring tests during execution in order to gather the dependencies between tests. They found that their proposed solution had a run time of less than one percent of the run time of the previous approaches to the problem. Their research shows that, when applied, their solution may increase the safety of other approaches which depend on selecting a subset of the total test suite.
Chapter 3

Approach

3.1 Novelty

Regression test selection approaches use transitive closure to select relevant tests. For any node in a directed graph, transitive closure is the set of all reachable nodes from the starting node. This thesis proposes a new approach to lessen the performance gap, by implementing a controlled limit to the transitive closure property. The aim of the limited transitive closure is to reduce the time taken to select tests, and to reduce the amount of superfluous tests selected.

The novelty of this approach lies in two aspects. The first novelty aspect is the research on the effects of limited transitive closure on class level regression test selection. Previous research on java programs has only considered regression test selection techniques which employ a full class firewall, as mentioned in section 2. This means that the set of tests considered for selection for each changed code class by these techniques is the full transitive closure of reachable classes in the dependency graph of a project.

The research in this paper adheres to the transitivity aspect of the firewall, but differs from previous research by ignoring the full closure aspect and rather considers a subset of the transitive closure set. This is done in order to attempt to reduce both the time taken to select tests, as well as the total time taken to execute tests.

This difference in transitive closure can be described as follows. In a graph which has a starting node n0, with neighbours n1 and n2, and n1 has neighbour n3, a transitive closure set is the set of all nodes. A limited transitive closure of e.g. level 1 will only reach the starting node
n0 at level 0, and its immediate neighbours n1 and n2 at level 1. This makes the set of the limited transitive closure nodes n0-n2. In order to extend the set to include node n3, the level limit would have to be incremented once, causing the limited transitivity to also include the neighbours of nodes n1 and n2.

The limited transitivity property works by employing the transitivity aspect a set amount of edges travelled away from the starting node during a breadth-first traversal, for each path available from each starting node. This approach of limited transitive closure is a heuristic approach, in that it has not been formally proven to outperform the standard approaches of static regression test selection. The approach of limited transitive closure in this research is done with the purpose of initial testing of the limited transitive closure to evaluate its effect on the performance of static regression test selection approaches.

The second novelty aspect is the comparison of multiple class level regression test selection techniques to each other. Legunsen et al. [1] has compared approach- and statement-level static regression test selection approaches to each other, as well as cross-level comparison of static regression test selection approaches, but class level regression test selection techniques has not been directly compared to each other before this study.

The static regression test selection software which implements the limited transitivity property is called HKRTS; Henrik Karlsson regression test selection. HKRTS is a standard static regression test selection approach with the one difference of adhering to the new limited transitive closure property instead of the standard transitive closure.

### 3.2 Finding Dependencies

#### 3.2.1 Gathering Dependencies

To gather the dependencies of the software project to be analyzed the jdeps tool was used. Jdeps is a standard library tool included in java from version 8 and forwards. Jdeps looks at a java class file and finds what other java class files that the target class file depends on.

The starting folder for which HKRTS searches is the containing folder of the project to be analyzed. HKRTS is made to recursively look through all files found in that folder, invoking the jdeps dependency finder for all java class files in the entire project.
When HKRTS has finished analyzing all files Jdeps will have created a .dot file for each class, in addition to the summary .dot file of the entire run. Each row of all of these files is then read, split, and curated into what is either a pair of classes where one class is dependent on the other, or a single class. A class which is dependent on the other class is called a dependor, and the class it is dependent on is called a dependency. If the line is a pair of dependor - dependency, the dependency relation is added into a map of dependencies. The reason for Jdeps occasionally putting a dependor without putting what it is dependent on is unclear, but as entries containing only one class does not provide any dependency data they are discarded.

While creating the pairs of dependors and dependencies the HKRTS software searches for classes from third-party libraries in the dependency relations. If either part of the dependency relation is a class from an external library, it is not added to the map of dependency relations. Classes from libraries outside of the project can be ignored without introducing any safety issues [1] for regression test selection approaches of class-level granularity. Excluding these libraries from the dependency graph will result in a smaller graph of the project dependencies. Reducing the size of the dependency graph will result in shorter run times and smaller resource costs when analyzing the dependencies in the graph.

### 3.2.2 Creating the Transitive Dependency Graph

The finished map of dependencies has a keyset of class names, which each returns a list of class names that are dependent on that class. Table 3.1 demonstrates an example of what a dependency map with four key nodes and four leaf nodes may look like.

<table>
<thead>
<tr>
<th>Keys</th>
<th>Dependent classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key1</td>
<td>Key2, leaf1, leaf2</td>
</tr>
<tr>
<td>Key2</td>
<td>Key3, key4, leaf1</td>
</tr>
<tr>
<td>Key3</td>
<td>Key4, leaf2, leaf3</td>
</tr>
<tr>
<td>Key4</td>
<td>leaf1, leaf3, leaf4</td>
</tr>
</tbody>
</table>

Table 3.1: Example of a dependency map.
is added to the graph.

Then for each node in the graph, the map is used to retrieve the list of all classes which depend on the class which the node represents. For each class in that list, a depth-first lookup [17] is made to see if it already existed in the graph, and therefore is a key in the map. Depth-first lookup was chosen since each node has two levels of connected nodes, but each level contains few nodes. This means that it is more
efficient to fully search all nodes reachable from a key before stepping to the next, rather than checking the bottom level for all keys first.

If the neighbour is a key it is added to the list of neighbours for the node being inspected, and if it is not a key in the map it is added to the graph as a leaf, stored as a plain String containing the name of the class which it represents. This process is shown in figure 3.1. As the leaves are dead ends in the graph they do not need to contain any information other than a class name.

This two-step node creation reduces time taken to create the graph. During the software setup, the time for HKRTS to create the dependency graph was several minutes for any of the target projects. This task was planned to take seconds, and after changing the approach of graph creation to the two-step node creation approach, the time taken for graph creation was reduced to seconds as planned.

3.3 Version Control

Software used for version control must be able to detect that changes has been made to code, in order to identify the differences between project revisions. For this experiment MD5 checksums are used as the approach of detecting code changes.

3.3.1 Checksum Verification

MD5 is an algorithm which has previously been used for cryptographic security, but is today considered broken and therefore obsolete within that field [9]. However this is not a problem for this project since the purpose the algorithm serves is to be capable of detecting that a change to specific java class files has occurred, without any concerns for cryptographic security.\footnote{The reason that MD5 checksum is considered obsolete is that it has a problem with collision. For our project MD5 is judged to be useable for the control of data integrity due to the fact that collision is unlikely to occur outside of an intentional attack.}

The way that MD5 is implemented into the HKRTS software is that any class file detected by HKRTS is read as a byte array via a file input stream. The read indata is then digested by the MessageDigest java library. The result is then convert into a HEX string, which is the
MD5 checksum for the file which can then be compared to the saved checksum on file.

### 3.3.2 Checksum Comparison

HKRTS keeps track of the previously read checksums in a checksum library, which is a local file. In order to find what classes are changed HKRTS compares the size of the current checksum library and the size of the new list of checksum input, and selects whichever is the smaller one. This is because once all checksums in the smaller list has been compared, the unchecked checksums in the larger list will be considered to have no counterpart in the smaller set.

Each checksum in the selected set is compared to the other set, along with the list of class names for each corresponding checksum. If there is a match of java class names and a mismatch of checksums, the class has been changed and is flagged for testing.

If a java class in the selected set does not exist in the other set, the situation is handled differently depending on which set was selected. If the checksum input was selected, the class is added to the library and is flagged for testing, since all new classes are considered untested. If the library was selected, a missing java class means that in the new revision the class has been removed from the project, and is removed from the checksum library as well.

In any revision pair the checksum library represents the old revision and the checksum input represents the new revision. In cases where the set sizes are equal HKRTS handles the data the same no matter which set is selected. This is handled by defaulting to selecting the library set.

The library is always considered to contain outdated checksums because it will always be at least one revision older than the checksum input. This means that the checksum input will always contain a more correct snapshot of the project for that revision.

### 3.4 Selecting Tests

When HKRTS detects a change of a java class file, it is marked as changed. When all checksums have been compared a complete list of all changed classes has been produced.
3.4.1 Identifying Dependencies

Each class on the list has direct dependencies and x-step away dependencies [11] [18]. Let the changed classes be \( c_0 \). A class \( c_1 \) which depends on \( c_0 \) is a direct dependency. A class \( c_2 \) which depends on \( c_1 \) has an indirect dependency of \( c_0 \), and is a 1-step away dependency of \( c_0 \). Any class \( c[i] \) that depends on \( c_2 \) has at least a 2-step away dependency of \( c_0 \), and is too far from the source \( c_0 \) to be selected for test running using the limited transitivity property with a level limit of 2.

The algorithm for selecting the tests can be described as follows. For any changed node \( c_0 \), find all dependors of \( c_0 \) which themselves have dependors, called key neighbours, and add them to list 1. Then find all dependors of \( c_0 \) which are leaves, add them to list 0. Then, for each node in list 1, find all dependors and add them to list 0. Finally, add list 1 to list 0. The add functions in this algorithm will only add unique items to each list. The algorithm is expressed in the pseudocode below. Figure 3.2 is a graphical representation of the difference in nodes selected by the limited transitivity property and the transitive closure property for a changed node \( n_2 \).

```plaintext
for changedClass in DependencyGraph
    find changedClass.neighbours
    listOfKeyNeighbours.add(neighbours)
    find changedClass.neighbourLeaves
```

Figure 3.2: Set generated by Transitive closure (left) and Limited Transitivity (right) for changed node \( n_2 \).
listOfNeighbours.add(neighborLeaves)

for neighbour in listOfKeyNeighbours
    find neighbour.neighbours
    find neighbour.neighbourLeaves
    listOfNeighbours.add(neighbours + neighbourLeaves)
    listOfNeighbours.add(listOfKeyNeighbours)

for class in listOfNeighbours
    if class isTest
        TestsToRun.add(class)

### 3.4.2 Running tests

In the set of selected classes, both ordinary java classes and java test classes are present. The list of selected relevant classes is filtered to only contain those classes which are java test classes. The list of relevant tests is passed to a test runner that executes the tests.

For this experiment Maven was used to execute tests, as Maven was the only test executor compatible with all three regression test selection approaches.

### 3.5 Implementation

A total of three regression test selection tools are used for the experiment, one dynamic approach and two static approaches.

The dynamic approach used is Ekstazi [32]. It has been integrated into several Apache projects, and is easy to use with Maven.

The first static approach is STARTS [33]. It is a static class-level regression test selection tool, and is the result of previous research in the area of static approaches of regression test selection.

The second static approach is the one introduced in with this thesis; HKRTS. It is available at

https://github.com/zZzebror/autotest-efficiency

The external library which HKRTS implements is the “Maven invoker” library, in order to use Maven to run the tests. It is available at

Chapter 4

Experiment Execution and Results

This chapter contains a description of the experiment set up, what data the experiments yielded and how it was gathered.

4.1 Protocol

The experiment protocol consists of the following six steps;

1. Select target projects to run the experiment on.

2. For each project, evaluate the last 100 revisions of the project in order to find which will produce meaningful experiment data.

3. Run the test selection approaches for each of the selected revisions.

4. Record the runtimes and number of tests selected by the softwares for each revision.

5. Repeat step 2 through 4 for all projects.

6. Summarize the data in ways that helps point to an answer to each research question.
4.1.1 Creating a Dataset

To compare the regression test selection approaches to each other the amount of tests selected, the time taken to select the tests, and the total runtime is recorded.

Additionally there needs to be some default data to draw comparisons from. An additional approach which simply runs all tests available in each revision is run as well. This approach has the purpose of recording the total number of tests in a revision, and the time taken to run them. This new approach is called AllTests.

All the regression test selection approaches needs to be compared to the AllTests approach as AllTests will be guaranteed to be as safe as possible, and the purpose of regression test selection is to decrease the runtime by increasing precision.

4.1.2 Selecting Projects

Open source projects were selected with the criteria:

1. Contains java class files
2. Large number of revisions
3. Publicly available on GitHub
4. Utilizes Maven

The criteria for a project to have a large number of revisions is to increase the amount of resulting data produced by the experiment. For any filtering process which removes data from a set and does not add to the set, increasing the maximum size of the data set which is to be filtered increases the maximum potential size of the set produced by the filtering process. For any set of code $X$, and for any function $f$ which selects a subset of indata, $f(X) \leq X$. This means that selecting a project which contains a small amount of revisions will at best result in a small amount of revisions selected for the experiment. This would not be ideal for this experiment since the purpose regression test selection solutions is to reduce the workload produced by testing a large number of revisions.

Selecting projects which contains a large number of revisions also ensures that the regression test selection solutions are run on a project of significant size, which contains frequent changes. This is the type of
project which will benefit the most from having a regression test selection solution implemented [7]. The time saved by regression test selection solutions is measured as a percentage of the time it takes to run all tests. This means that regression test selection solutions have greater impact on projects where running all tests take longer. Therefore, the type of projects that benefit the most from regression test selection are those where running all tests take longer, and projects where the testing process is run more frequently.

4.1.3 Filtering Revisions

The reason that the number of revisions considered for each project is 100 is because previous research [1] has proved that using 100 revisions results in a usable data set size. In order for a project revision to be used in the experiment, it has to fulfill the following criteria:

1. The revision has to contain at least one change to at least one java class file
2. The revision has to compile successfully
3. The AllTests approach has to run to completion
4. The Ekstazi approach has to run to completion
5. The HKRTS approach has to run to completion
6. The STARTS approach has to run to completion

If a project compiles successfully, there still may be errors that causes the tests to break during runtime. This could be caused either by the project breaking during the run before executing the tests, or there may be classes that contain errors severe enough that they interrupt the run during the test execution. If any of the approaches do not run to completion due to these reasons they will output an error instead of test data, and the revision is discarded. In order to assure that the sample size is equal for all four approaches used in the experiment, the data produced by the other approaches for that run is discarded as well.

The reason for the first criteria is that all three regression test selection approaches detects changes only in java class files. Non-Java revisions will therefore not produce any data for this experiment. For
this reason, if a revision does not contain changes to any java class it is
discarded.

4.1.4 Running Experiment Software

The first approach is AllTests, which runs all class files in the project
that Maven defines as tests\(^1\). This will give a measurement of the total
number of test classes in the project, as well as a baseline run time
which the total run time of any regression test selection approach have
to beat. If a regression test selection solution has a longer run time
than just running all tests, it is useless for the purpose of regression
test selection.

The second approach is the Dynamic regression test selection soft-
ware Ekstazi\(^2\).

The third approach is the STARTS software\(^3\).

The fourth approach is our own solution, which is a static regres-
sion test selection approach called HKRTS (Henrik Karlsson regression
test selection)\(^4\).

4.1.5 Recording Results

For each revision the data is recorded after running the four approaches.

For AllTests, the runtime is saved, along with the number of tests
that exists for the current revision of the project.

For Ekstazi, STARTS, and HKRTS, the number of tests selected is
saved, in addition to the time to find the subset of tests to run, and the
time to run the selected tests.

4.1.6 Summarizing Data

The run time data is not deterministic due to aspects like background
processes on the computer taking up some computational power, but
repeated runs for the same revision will produce similar run times.
The amount of tests in the test suite is deterministic as the complete
test suite is known before running.

\(^1\)Executed with the "mvn test" command
\(^2\)It is executed with the command “mvn ekstazi:ekstazi”.
\(^3\)STARTS was ran with the command "mvn starts:starts -DstartsLogging=FINE".
\(^4\)The command to run HKRTS is the software name and the directory of the
project which was being subject to the experiment.
The data the three regression test selection approaches produce are the total runtime, the time taken to select tests, and the number of tests chosen to run. The run time data is not deterministic. As the regression test selection methods select tests based on dependencies found in the revision analyzed, the amount of tests chosen is deterministic as the same dependencies will be found each time the revision is run.

The revisions of each project are set into \([r(i), r(i+1)]\) is the previous version which fulfills the criteria for a successfully run revision and \(r(i+1)\) is the current version which fulfills the criteria. For each such pair of revisions in a project, the data for the different approaches are gathered. This data is entered into a table on the row which is named after the \(r(i+1)\) revision. Because the 100 revisions considered for each project are filtered through the criteria for including a revision the data for each project will contain different amounts of revision pairs.

The data will be summarized into tables and figures with the purpose of attempting to answer the research questions.

For research question 1, number of tests selected, a table is created which shows tests selected by each regression test selection approach relative to AllTests. This figure represents how large of a part of all tests in the suite that each approach chooses to run. The metric which is used here is the total amount of tests run by each approach, compared to the amount of tests run by AllTests.

For research question 2, runtime, a table is created which shows the absolute end-to-end run time for the three solutions. The numbers presented in this table are the end-to-end runtimes of each approach, compared to the total runtime for AllTests for each revision pair of each project.

This table shows the minimum, maximum, and average differences of run time between AllTests and Ekstazi, STARTS, and HKRTS, for each project in the experiment.

In order for a regression test selection solution to be considered safe, all tests which may detect faults have to be selected. In addition to this, in order for a regression test selection solution to be considered precise, it needs to select to run only tests which may detect faults [1]. This means that in order for a regression test selection solution to be both completely safe and completely precise, it has to select a subset of all tests that contains all of the tests which may detect faults and only those tests.

There is no way to find which tests will detect faults for any given
revision without running the tests, since testing is the way to find what tests detect faults. However it is possible to compare safety and precision performances between two regression test selection methods, by using formulas [1] which are based on the tests selected by each approach. The result of these formulas are expressions in percentages of how much more or less safe and precise one approach is compared to another.

The number of tests selected is used to calculate the values of precision and safety of STARTS and HKRTS. Since precision and safety are relative values, the static approaches will be compared to Ekstazi when calculating the values. This lets us make a comparison of effective performance between the two static regression test selection approaches and evaluate which of the approaches performance is closest to that of Ekstazi.

For research question 3, relative safety, a table is created which shows the relative safety of STARTS and HKRTS as compared to Ekstazi. The difference in safety is measured as a function of the difference in tests selected to run by two regression test selection solutions. To find the rate of safety violation, let E be the set of tests chosen to run by Ekstazi, and let T be the set of tests chosen to run by the approach for which to calculate the safety. To calculate the safety the formula \((E/T)/(E \cup T)\) is used [1]. The resulting value is an expression in percentages of how much more or less safe than Ekstazi the compared static approach is.

For research question 4, relative precision, a similar table is created. The main difference between measuring the safety and the precision is that calculating the precision uses a similar, but different formula. The formula uses the same denominated metrics, but flips the division of the sets of tests. The formula is \((T/E)/(E \cup T)\) [1], and the result is an expression in percentages of how much more or less precise the static approaches are compared to Ekstazi.

### 4.2 Approach

The evaluation protocol for the experiment contains three different approaches of regression test selection which are run on revisions of one project at a time. The three approaches are run for revisions of the project that build successfully. Revisions for which all three ap-
proaches finishes a full test suite run data will be collected, until a data set for the project has been created.

4.3 Running the Experiment

For this experiment 23 different projects were chosen to collect experimental data from. One project came from the company Cinnobar AB, and the other projects are open source projects found on GitHub.

4.3.1 Projects Chosen

From Cinnobar AB the TE-proteus project was chosen, which is a snapshot. Using a snapshot instead of a regular version does not impact the experiment since the differences are that a snapshot has certain software arguments set to default rather than being able to be set by the user. This difference is covered by the project test cases and does not need to be considered further for this experiment. For the sake of the experiment, running a project with the default settings will still qualify each revision within the four criteria. All three regression test selection approaches are easier to set up working correctly if there are less special inputs that have to be satisfied in order to run the tests.

Also chosen for data collecting were the open source projects. These projects were chosen for being the first of the projects considered which passed all criteria previously discussed in chapter 4.1.2.

4.3.2 Physical Setup for Experiment Run

The experiment was run on a Intel(R) Core(TM) i5-4690K CPU machine with 12GB of RAM, running Windows 7 64-bit version. On this physical machine a virtual machine was set up in order to run the experiment. The virtual machine ran on Ubuntu 18.04. The virtual machine was using 2048MB of memory, 2 CPUs, and 128MB of video memory; everything else was the same as the host machine. The virtual machine ran Oracle Java, 64-Bit version 1.8.0 161.
4.4 Results

4.4.1 Metrics

Run time
The time from entering the test execution command for the software until the software has finished the complete process execution.

Tests
The number of tests each regression test selection approach selects to run.

Safety
Safety is a relative measurement for how much more or less safe one regression test selection approach is when compared to another regression test selection approach.

Precision
As mentioned in section 2.4, a regression test selection approach which selects only tests which are affected by code changes is completely precise. Precision is a relative measurement for how much more or less precise one regression test selection approach is when compared to another regression test selection approach. Violations of precision are calculated with the same base values as safety, but uses a different formula.

4.4.2 Dataset
The dataset consists of the run times, time to select tests, and the number of tests selected by the three regression test selection approaches when run on the revisions of 19 software projects. For this experiment, a total of 23 projects were considered approved according to the criteria. Due to problems which arose during the running of the experiment, the projects compile-testing, commons-compress, jankotek.mapdb, and TE-proteus were discarded. The process of discarding of these projects is expanded upon further in section 4.5.
### Table 4.1: The projects used for the experiment.

<table>
<thead>
<tr>
<th>ID</th>
<th>Project name</th>
<th>Initial Rev SHA</th>
<th>REVS</th>
<th>Tests</th>
<th>T[S] RunAll</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>invokebinder</td>
<td>896b4c2a</td>
<td>15</td>
<td>109.3</td>
<td>3.3</td>
</tr>
<tr>
<td>p2</td>
<td>logback-encoder</td>
<td>00ab5ca9</td>
<td>28</td>
<td>255.4</td>
<td>40.8</td>
</tr>
<tr>
<td>p3</td>
<td>commons-cli</td>
<td>085a1538</td>
<td>22</td>
<td>388</td>
<td>23</td>
</tr>
<tr>
<td>p4</td>
<td>commons-dbutils</td>
<td>08e066ef</td>
<td>25</td>
<td>263.7</td>
<td>9.8</td>
</tr>
<tr>
<td>p5</td>
<td>commons-fileupload</td>
<td>b1498e98</td>
<td>12</td>
<td>76.8</td>
<td>9.7</td>
</tr>
<tr>
<td>p6</td>
<td>commons-validator</td>
<td>e4a6a3ca</td>
<td>23</td>
<td>466.9</td>
<td>23.3</td>
</tr>
<tr>
<td>p7</td>
<td>asterisk-java</td>
<td>8690bd05</td>
<td>25</td>
<td>240.7</td>
<td>29.4</td>
</tr>
<tr>
<td>p8</td>
<td>commons-codec</td>
<td>c8c94f9a</td>
<td>19</td>
<td>881.8</td>
<td>24.2</td>
</tr>
<tr>
<td>p9</td>
<td>commons-email</td>
<td>19074718</td>
<td>15</td>
<td>185.3</td>
<td>24.4</td>
</tr>
<tr>
<td>p10</td>
<td>commons-collections</td>
<td>8b66a577</td>
<td>20</td>
<td>24566.7</td>
<td>39.3</td>
</tr>
<tr>
<td>p11</td>
<td>commons-lang</td>
<td>2a116425</td>
<td>19</td>
<td>4344.2</td>
<td>42.2</td>
</tr>
<tr>
<td>p12</td>
<td>commons-imaging</td>
<td>84373011</td>
<td>24</td>
<td>507.5</td>
<td>44.9</td>
</tr>
<tr>
<td>p13</td>
<td>commons-dbcp</td>
<td>63bee061</td>
<td>20</td>
<td>951</td>
<td>110</td>
</tr>
<tr>
<td>p14</td>
<td>b.HikariCP</td>
<td>a5bd6c0a</td>
<td>14</td>
<td>130.8</td>
<td>183</td>
</tr>
<tr>
<td>p15</td>
<td>commons-io</td>
<td>8d5c46f4</td>
<td>17</td>
<td>1331.9</td>
<td>144</td>
</tr>
<tr>
<td>p16</td>
<td>addthis.stream-lib</td>
<td>8d46dd73</td>
<td>11</td>
<td>150.6</td>
<td>147</td>
</tr>
<tr>
<td>p17</td>
<td>commons-math</td>
<td>3c0e9d83</td>
<td>8</td>
<td>4099.4</td>
<td>145.4</td>
</tr>
<tr>
<td>p18</td>
<td>OpenTripPlanner</td>
<td>71450d42</td>
<td>16</td>
<td>74.9</td>
<td>50</td>
</tr>
<tr>
<td>p19</td>
<td>commons-pool</td>
<td>12ba9290</td>
<td>7</td>
<td>280</td>
<td>325.9</td>
</tr>
<tr>
<td></td>
<td>Average</td>
<td></td>
<td>17.9</td>
<td>2068.7</td>
<td>74.7</td>
</tr>
</tbody>
</table>

#### 4.4.3 Empirical Results

Table 4.1 provides an overview of the projects used in the experiment, their revisions, and the average number of tests and runtime for AllTests. The SHA value is the SHA-checksum of the revision for which the experiment started this project on, meaning that, if the initial revision pair is \([r(0), r(1)]\), the Initial Rev SHA is that of \(r(0)\). These SHA checksums are generated automatically by GitHub when a software revision is uploaded and act as identifiers for the project revisions. These SHA checksums do not affect the project files, which is why HKRTS uses the previously mentioned MD5 checksums for version control.

REVS is the number of revision pairs as explained earlier, where \((i)\) is a revision number. This number denominates the amount of revisions out of the 100 considered which passed the criteria and thus was used in the experiment. Since this number is out of 100, it is also exactly the percentage of revisions which passed the criteria for each
Tests is the average number of tests across all revisions for each project.

$T[S]$ RunAll is the average runtime for the AllTests approach for each revision in the project measured in seconds.

**Research Question 1: Number of Tests**

Table 4.2 represents how large of a part of all tests in the suite that Ekstazi, STARTS, and HKRTS chooses to run, respectively. The numbers in the table are percentage values, and represent how large a portion of all tests in the test suite were selected to run by each approach. The Average column is the average value for each approach for all projects in the experiment.

Selecting fewer tests than AllTests can have positive and negative effects. The potential upside from selecting fewer tests is the fact that running fewer tests should take less time. In contrast to this, the downside of selecting fewer tests is that it decreases the safety.

The data presented in table 4.2 shows that for each project, STARTS selects the most tests of the three approaches. HKRTS selects the least tests, and in the case of project commons-math HKRTS selects 0 tests on average. The most likely reason for HKRTS selecting 0 tests is that for the revisions used in the experiment, no changed class had a test class reachable in the dependency graph under the limited transitivity property.

**Research question 2: Runtime**

Table 4.3 presents a comparison in percentages of how much the runtimes of each of the approaches take compared to the AllTests approach. The min, max, and average values are the lowest percentage, highest percentage, and overall average percentage of the AllTests runtime taken per project by each approach.

The data in this table shows that on average each of the three approaches take less time to run than AllTests. However each of the approaches have multiple projects where they average a runtime higher than that of AllTests, and for the invokebinder project all three approaches had a minimum runtime which was higher than the AllTests runtime. invokebinder has the lowest average AllTests runtime at 3.3 seconds,
<table>
<thead>
<tr>
<th>Project</th>
<th>Ekstazi tests</th>
<th>STARTS tests</th>
<th>HKRTS tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>80.3</td>
<td>87</td>
<td>3.7</td>
</tr>
<tr>
<td>p2</td>
<td>28.6</td>
<td>44.3</td>
<td>4.3</td>
</tr>
<tr>
<td>p3</td>
<td>30.9</td>
<td>51.3</td>
<td>3.4</td>
</tr>
<tr>
<td>p4</td>
<td>34.5</td>
<td>46.9</td>
<td>3.2</td>
</tr>
<tr>
<td>p5</td>
<td>55.4</td>
<td>62.6</td>
<td>8.6</td>
</tr>
<tr>
<td>p6</td>
<td>12.9</td>
<td>18.7</td>
<td>1</td>
</tr>
<tr>
<td>p7</td>
<td>17.7</td>
<td>26.2</td>
<td>3</td>
</tr>
<tr>
<td>p8</td>
<td>14.7</td>
<td>32.9</td>
<td>0.9</td>
</tr>
<tr>
<td>p9</td>
<td>69.8</td>
<td>86</td>
<td>4.8</td>
</tr>
<tr>
<td>p10</td>
<td>28.4</td>
<td>51.2</td>
<td>0.2</td>
</tr>
<tr>
<td>p11</td>
<td>48.5</td>
<td>54.2</td>
<td>1.7</td>
</tr>
<tr>
<td>p12</td>
<td>46.2</td>
<td>75.9</td>
<td>4.4</td>
</tr>
<tr>
<td>p13</td>
<td>61.9</td>
<td>67.8</td>
<td>2</td>
</tr>
<tr>
<td>p14</td>
<td>84.4</td>
<td>94.5</td>
<td>21.6</td>
</tr>
<tr>
<td>p15</td>
<td>26.3</td>
<td>44.5</td>
<td>1.7</td>
</tr>
<tr>
<td>p16</td>
<td>30.7</td>
<td>46.7</td>
<td>4.9</td>
</tr>
<tr>
<td>p17</td>
<td>38.8</td>
<td>52.8</td>
<td>0</td>
</tr>
<tr>
<td>p18</td>
<td>28.5</td>
<td>65.3</td>
<td>70.2</td>
</tr>
<tr>
<td>p19</td>
<td>75</td>
<td>95.1</td>
<td>14.7</td>
</tr>
<tr>
<td>Average</td>
<td>42.8</td>
<td>58.1</td>
<td>8.1</td>
</tr>
</tbody>
</table>

Table 4.2: Average percentage of tests selected relative to AllTests.
### Table 4.3: End-to-end test time relative to AllTests.

<table>
<thead>
<tr>
<th>Project</th>
<th>Ekstazi runtime</th>
<th>STARTS runtime</th>
<th>HKRTS runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Min (p1)</td>
<td>Max (p16)</td>
<td>Average (p15)</td>
</tr>
<tr>
<td>p1</td>
<td>104.9</td>
<td>168</td>
<td>129.9</td>
</tr>
<tr>
<td>p2</td>
<td>2</td>
<td>722.9</td>
<td>124.4</td>
</tr>
<tr>
<td>p3</td>
<td>7.8</td>
<td>641.4</td>
<td>173.6</td>
</tr>
<tr>
<td>p4</td>
<td>56.3</td>
<td>203.8</td>
<td>108.6</td>
</tr>
<tr>
<td>p5</td>
<td>74</td>
<td>322.4</td>
<td>129.8</td>
</tr>
<tr>
<td>p6</td>
<td>13</td>
<td>98</td>
<td>70.6</td>
</tr>
<tr>
<td>p7</td>
<td>7.6</td>
<td>221.6</td>
<td>47.1</td>
</tr>
<tr>
<td>p8</td>
<td>23.2</td>
<td>117.7</td>
<td>46.2</td>
</tr>
<tr>
<td>p9</td>
<td>48.3</td>
<td>126</td>
<td>90</td>
</tr>
<tr>
<td>p10</td>
<td>22</td>
<td>106.4</td>
<td>50.2</td>
</tr>
<tr>
<td>p11</td>
<td>32.8</td>
<td>110</td>
<td>72.3</td>
</tr>
<tr>
<td>p12</td>
<td>28.9</td>
<td>133.7</td>
<td>85.3</td>
</tr>
<tr>
<td>p13</td>
<td>13.8</td>
<td>103.9</td>
<td>75</td>
</tr>
<tr>
<td>p14</td>
<td>92.2</td>
<td>105.5</td>
<td>100</td>
</tr>
<tr>
<td>p15</td>
<td>3.9</td>
<td>104.7</td>
<td>31</td>
</tr>
<tr>
<td>p16</td>
<td>3.3</td>
<td>105.8</td>
<td>44.6</td>
</tr>
<tr>
<td>p17</td>
<td>23.7</td>
<td>143.4</td>
<td>70.6</td>
</tr>
<tr>
<td>p18</td>
<td>40.3</td>
<td>303.4</td>
<td>149</td>
</tr>
<tr>
<td>p19</td>
<td>97.8</td>
<td>99.4</td>
<td>78.9</td>
</tr>
<tr>
<td>Average</td>
<td>37.1</td>
<td>210.4</td>
<td>39</td>
</tr>
</tbody>
</table>

### Table 4.4: Portion of selection time relative to total runtime for each approach.

<table>
<thead>
<tr>
<th>Project</th>
<th>Ekstazi</th>
<th>STARTS</th>
<th>HKRTS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Min (p1)</td>
<td>Max (p16)</td>
<td>Average (p15)</td>
</tr>
<tr>
<td>p1</td>
<td>0.1</td>
<td>14.9</td>
<td>2.7</td>
</tr>
<tr>
<td>p2</td>
<td>0</td>
<td>24.2</td>
<td>7.5</td>
</tr>
<tr>
<td>p3</td>
<td>1.4</td>
<td>0</td>
<td>0.8</td>
</tr>
<tr>
<td>p4</td>
<td>0</td>
<td>3.3</td>
<td>1.1</td>
</tr>
<tr>
<td>p5</td>
<td>0</td>
<td>1.5</td>
<td>0.9</td>
</tr>
<tr>
<td>p6</td>
<td>0</td>
<td>3.6</td>
<td>2.5</td>
</tr>
<tr>
<td>p7</td>
<td>0</td>
<td>6.3</td>
<td>2.8</td>
</tr>
<tr>
<td>p8</td>
<td>0</td>
<td>3.6</td>
<td>2.4</td>
</tr>
<tr>
<td>p9</td>
<td>0</td>
<td>3.1</td>
<td>1.4</td>
</tr>
<tr>
<td>p10</td>
<td>0</td>
<td>4.8</td>
<td>2.8</td>
</tr>
<tr>
<td>p11</td>
<td>0</td>
<td>5.4</td>
<td>2.5</td>
</tr>
<tr>
<td>p12</td>
<td>0</td>
<td>3.3</td>
<td>1.2</td>
</tr>
<tr>
<td>p13</td>
<td>0</td>
<td>0.7</td>
<td>1</td>
</tr>
<tr>
<td>p14</td>
<td>0</td>
<td>0.6</td>
<td>0.5</td>
</tr>
<tr>
<td>p15</td>
<td>0</td>
<td>3.4</td>
<td>1.3</td>
</tr>
<tr>
<td>p16</td>
<td>0</td>
<td>0.8</td>
<td>0.4</td>
</tr>
<tr>
<td>p17</td>
<td>0</td>
<td>2.9</td>
<td>1.5</td>
</tr>
<tr>
<td>p18</td>
<td>0</td>
<td>3.3</td>
<td>1.6</td>
</tr>
<tr>
<td>p19</td>
<td>0</td>
<td>0</td>
<td>0.1</td>
</tr>
<tr>
<td>Average</td>
<td>0.1</td>
<td>4.8</td>
<td>1.8</td>
</tr>
</tbody>
</table>
causing any attempt optimize the test suite to take longer than just running all tests.

For the projects where AllTests had its highest runtimes, each of the three regression test selection approaches ran faster than AllTests, both when observing the average revision values of the approaches as well as when cherry-picking the revisions where each approach performed the worst.

Table 4.4 demonstrates how the average runtime of each approach was divided between selecting tests and running tests for each project.

The data presented in table 4.4 shows that for the Ekstazi and STARTS approaches, selecting tests took less than 10 percent of total runtime on average. In contrast to this, HKRTS took over half of total runtime to select tests on average, with the only project where HKRTS averaged less than half of the runtime for test selection being the logback-logstash project. This could be the effect of HKRTS selecting fewer tests than Ekstazi and STARTS, causing HKRTS to have lower total runtime in comparison. If different approaches have similar selection times, the approach which has a higher absolute runtime will have a lower percentage of total time taken to select tests.

Research Question 3: Safety

Table 4.5 represents the relative Safety of STARTS and HKRTS when compared to Ekstazi, expressed as percentages. Lower percentages are better, since that implies that the performance of STARTS and HKRTS is closer to that of Ekstazi.

For all projects, in any revision where at least one of the two approaches chose to run no tests, such a revision is counted as a 0. This is due to the fact that safety is a relative measurement, and anything non-zero compared to a 0 differs by infinity percent. Thus they are counted as 0 for the sake of simplicity. This is the reason for the minimum value of 0 for many of the projects.

The data presented in this table shows that STARTS has an average safety loss of 1.2 percent when compared to Ekstazi, meaning that in the average case there is almost no difference between the two approaches. In the worst case scenario, in this experiment projects asterisk-java and commons-imaging, STARTS is 50 percent less safe than Ekstazi at worst.

In contrast to this, HKRTS compared to Ekstazi gives an average
<table>
<thead>
<tr>
<th>Project</th>
<th>STARTS safety</th>
<th>HKRTS safety</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Min</td>
<td>Max</td>
</tr>
<tr>
<td>p1</td>
<td>0.3</td>
<td>5.6</td>
</tr>
<tr>
<td>p2</td>
<td>0</td>
<td>3.1</td>
</tr>
<tr>
<td>p3</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>p4</td>
<td>0</td>
<td>23.8</td>
</tr>
<tr>
<td>p5</td>
<td>0.5</td>
<td>2.1</td>
</tr>
<tr>
<td>p6</td>
<td>0</td>
<td>6.3</td>
</tr>
<tr>
<td>p7</td>
<td>0</td>
<td>50</td>
</tr>
<tr>
<td>p8</td>
<td>0</td>
<td>4.5</td>
</tr>
<tr>
<td>p9</td>
<td>0</td>
<td>0.8</td>
</tr>
<tr>
<td>p10</td>
<td>0</td>
<td>1.7</td>
</tr>
<tr>
<td>p11</td>
<td>0</td>
<td>5.5</td>
</tr>
<tr>
<td>p12</td>
<td>0</td>
<td>50</td>
</tr>
<tr>
<td>p13</td>
<td>0</td>
<td>0.4</td>
</tr>
<tr>
<td>p14</td>
<td>0.3</td>
<td>0.4</td>
</tr>
<tr>
<td>p15</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>p16</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>p17</td>
<td>0</td>
<td>0.3</td>
</tr>
<tr>
<td>p18</td>
<td>0</td>
<td>1.4</td>
</tr>
<tr>
<td>p19</td>
<td>0.1</td>
<td>0.2</td>
</tr>
<tr>
<td>Average</td>
<td>0.1</td>
<td>10.3</td>
</tr>
</tbody>
</table>

Table 4.5: Relative safety of STARTS and HKRTS.
CHAPTER 4. EXPERIMENT EXECUTION AND RESULTS

Table 4.6: Relative precision of STARTS and HKRTS.

<table>
<thead>
<tr>
<th>Project</th>
<th>STARTS precision</th>
<th>HKRTS precision</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Min</td>
<td>Max</td>
</tr>
<tr>
<td>p1</td>
<td>0.4</td>
<td>5.6</td>
</tr>
<tr>
<td>p2</td>
<td>0.2</td>
<td>32.9</td>
</tr>
<tr>
<td>p3</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>p4</td>
<td>0</td>
<td>31.3</td>
</tr>
<tr>
<td>p5</td>
<td>0.6</td>
<td>2.1</td>
</tr>
<tr>
<td>p6</td>
<td>0.1</td>
<td>6.3</td>
</tr>
<tr>
<td>p7</td>
<td>0</td>
<td>75</td>
</tr>
<tr>
<td>p8</td>
<td>0</td>
<td>7.6</td>
</tr>
<tr>
<td>p9</td>
<td>0.3</td>
<td>32.8</td>
</tr>
<tr>
<td>p10</td>
<td>0</td>
<td>4.5</td>
</tr>
<tr>
<td>p11</td>
<td>0</td>
<td>1.5</td>
</tr>
<tr>
<td>p12</td>
<td>0.1</td>
<td>50</td>
</tr>
<tr>
<td>p13</td>
<td>0</td>
<td>49.8</td>
</tr>
<tr>
<td>p14</td>
<td>0.4</td>
<td>0.7</td>
</tr>
<tr>
<td>p15</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>p16</td>
<td>0.4</td>
<td>24.3</td>
</tr>
<tr>
<td>p17</td>
<td>0</td>
<td>0.5</td>
</tr>
<tr>
<td>p18</td>
<td>0</td>
<td>31.5</td>
</tr>
<tr>
<td>p19</td>
<td>0.2</td>
<td>0.3</td>
</tr>
<tr>
<td>Average</td>
<td>0.1</td>
<td>20.6</td>
</tr>
</tbody>
</table>

safety loss of 17.4 percent. The project where HKRTS performs the worst is commons-imaging, where HKRTS has a safety loss of 99.7 percent, meaning it will only detect roughly 0.3 percent of the faults which Ekstazi detects.

Research Question 4: Precision

Table 4.6 represents the relative Precision of STARTS and HKRTS when compared to Ekstazi, expressed as percentages. Lower percentages are better, since that implies that the performance of the approach is closer to that of Ekstazi.

As was done for the question of safety, any revision where at least one of the two approaches chose to run no tests is counted as a 0. Since safety and precision are relative measurements, and anything non-zero compared to a 0 differs by infinity percent. Thus they are counted as 0 for the sake of simplicity, which as for safety is the reason for the minimum value of 0 for many of the projects.

The data in this table shows that STARTS averages a precision loss
of 2.5 percent, with the worst case performance being project asterisk-java, where STARTS has a precision loss of 75 percent.

HKRTS averages a precision loss of 1.1 percent. The worst performance of HKRTS was for project asterisk-java, where it had a 80 percent precision loss.

Compared to each other, HKRTS has less than half of the precision loss that STARTS does. HKRTS also has a lower average precision loss for each individual project except for commons-codec and OpenTripPlanner, where it was equal, and higher, respectively. In contrast to this, HKRTS worst performance still resulted in a higher precision loss than STARTS worst performance.

4.5 Problems Encountered During Experiment Run

In total there were 23 projects, of which 19 ran successfully and had its data recorded. I was unable to gather data for 4 of the projects chosen.

The first problematic project was compile-testing. Ekstazi did not work “out of the box” for any revision of the project. Various attempts at solving this problem were made but none resulted in a working configuration. Since there would be no data to compare the static approaches with, this project was discarded for the results of the experiment run.

The second project was commons-compress. The problem with this project was that each revision took the greater part of one hour (with an average of just above 40 minutes) to run all of the approaches. The decision to skip this project for the run of the experiment was due to the serious strain on the schedule it would impose, and the potential gain for including it was not deemed to be worth it for the experiment.

The third project for which I were unable to gather data was jankotek.mapdb. The problem encountered in the running of this project was that there was only one revision for which AllTests ran at all, and for that one revision Ekstazi did not run. For this reason the project was discarded, since regression test selection needs at least one pair of revisions in order to function.

There were multiple factors which affected the experiment run negatively when attempting to run the test approach on the TE-proteus project.
The software transition from Maven, which was used by all of the open source projects, to Gradle which was used in TE-proteus took some time to implement. This was a violation of the criteria set up for the project selection, and thus reason for discarding.

The only saved older versions of the project which existed was finished release versions, which limited the amount of revisions available. This was again further reduced by the fact that even if older versions of the project existed, there were decrepit dependencies which did not. In the end only four revisions existed (9.13.0 to 9.15.0) for the new platform which also had recent enough dependencies to run. One older revision (5.9.1) of a previous setup also existed and ran successfully.

In the end, only three revisions in total ran at all, with only two of them being able to gather complete information about Ekstazi (version 9.13.0 would not yield information of the test run, due to what is deemed a software compatibility error). This brings the only “useful” revision pair to consist of one 4-year-old version of the project, and the most up-to-date version of the project. This was deemed to big of a change in project functionality between revisions to realistically use regression test selection since every single java class would be unrecognizable, and therefore flagged for testing. For this reason, the TE-proteus is not included in the data of the experiment.
Chapter 5

Discussion

5.1 Tests Selected

HKRTS selecting less than a tenth of all tests would be beneficial for those revisions which there are few code changes. This is because changing a smaller part of the software code will result in less of the software code being affected by the changes. The revisions with these small amounts of changes are likely those where the software developer has acknowledged that there is less risk than usual, and so a regression test selection method which runs fewer tests but completes the testing process faster will be suitable. HKRTS’ limited transitivity property would also be beneficial for the projects where each class change only affects the classes close to it in the software system.

Developers on a project using a regression test selection approach like HKRTS would expect less downtime when running their tests for integration, potentially increasing their overall workplace effectivity. They would also expect a higher number of bugs going undetected compared to when the project had chosen to use a regression test selection like Ekstazi, due to the implication of less test coverage equaling fewer faults detected.

HKRTS will likely not become a replacement of current [2] [3] [6] [7] [8] approaches because it is too unsafe by comparison. However it may be used alongside them, as a complement to run instead when there are few changes made for a revision. Reducing the time taken to integrate code into a software project would in turn shorten the end to end time taken to produce software, which would be economically beneficial in the short term as it would allow for a shorter time be-
between the start of the project and the first sellable software.

HKRTS could affect the lack of manual testing in a positive way; if the process of running a test suite takes less time developers may be more likely to do it [4] [5]. In the cases where developers are working on software projects which do not have an automatic system for running test suites set up developers will have to run the test suite manually, which will interrupt their work flow as they have to wait for the running of the suite to finish. HKRTS reducing the time taken to run the tests may potentially reduce the impact this interruption has.

5.2  Runtime

5.2.1  Runtime Compared to AllTests

Table 4.3 shows that the shorter the AllTests runtime is for a project revision, the higher the rate is at which one of the approaches runtime exceed AllTests runtime.

Combining the data in table 4.3 with the data in table 4.1 shows that when the AllTests runtime is exceeded by one of the selection approaches the percentage of the AllTests runtime exceeded is lower for those projects where AllTests had a higher average runtime. This, combined with similar results of previous research [1], suggests that the exceeded runtime is because of the test selection process rather than the test running process, as running a subset of tests can not take longer than running the full set.

If a regression test selection approach were to save a certain percentage of AllTests runtime, the effect of that approaches runtime savings would become increasingly noticeable for a human user as AllTests runtime grows higher.

If a regression test selection approach exceeds AllTests runtime at short runtimes but has a shorter runtime than AllTests at long runtimes the effects of the approach may be a net time gain, depending on how many revisions where AllTests have long runtimes and how many have shorter runtimes.

5.2.2  Test Selection Portion

Table 4.4 shows that higher average runtimes more often result in lower average percent of total runtime taken for test selection, and lower av-
average runtimes more often result in a higher test selection time percentage.

Observing the data for STARTS shows that it has an average test selection percentage of 7.4, in contrast to the average 64.5 percentage of total run time as selection time for HKRTS, due to HKRTS selecting to run fewer tests and thus achieving a lower test runtime.

There could be noticeable effects for users of a regression test selection approach with a high test selection time percentage. If the selection algorithm is slow and inefficient it will have a negative impact on the performance of the regression test selection approach, as the purpose of regression test selection approaches is to reduce the time taken for testing [16].

5.3 Safety

When comparing both static approaches against Ekstazi, the average safety loss of HKRTS is 14.5 times higher than that of the STARTS approach. When comparing the worst performances of the static approaches against each other, HKRTS has a more than four times larger safety loss.

The impact of the safety of a approach such as HKRTS is negative. The safety losses have similar effects as those discussed for the data of selected tests, in that it makes HKRTS unable to be a stand-alone solution. For HKRTS to be able to be a viable alternative to other approaches it would have to be combined with AllTests in order to prevent a high number of software faults going undetected. When to run HKRTS and when to run AllTests would be up to the developer, which has been proven to be unsafe [4] [5].

HKRTS would not be a suitable choice for projects which prioritize fault detection as the risk of faults going undetected rises as the safety decreases. These projects would have nothing to gain from using HKRTS and would be better of using another approach like STARTS, Ekstazi, or even AllTests.

5.4 Precision

The effects of an approach with higher precision are that software tested using such approaches will have their test suites run in a shorter
amount of time, due to less superfluous tests being ran. This will in
turn increase the overall efficiency of developers working on projects
utilizing these kinds of regression test selection approaches. If the ap-
proach is widespread enough, it may have an effect on the industry in
that it may increase the expected and actual efficiency of developers,
shortening the time taken to complete software development projects.

Due to selecting very few superfluous tests HKRTS will reduce the
time taken to run the test suites when compared to AllTests, as well as
to STARTS. The combination of this precision with the previously men-
tioned safety losses makes HKRTS ideal to run on smaller revisions, as
those revisions will benefit from the precision while being less affected
by the negative effects of the safety loss of HKRTS. Smaller revisions
will suffer less because revisions with fewer changed classes will affect
less parts of the software project, causing a larger part of the affected
classes in the dependency graph to be within the limit level detected
by the limited transitivity property of HKRTS.
Chapter 6

Threats to Validity

6.1 Threats to Internal Validity

The data point of run time for each approach could be a source of a threat to validity. As the run times are factors of computations in order to both find and run tests, anything outside the scope of the experiment which may affect the CPU is a potential threat to validity.

Since there was no monitoring of the CPU load during the experiment it is not possible to prove that no such effect on the CPU occurred. However, nothing was found in the results which points to a heavy CPU load affecting any one approach, project, or revision in particular. That is to say, if such an effect was present then it was not observable within the data, as it likely affected each of the approaches evenly.

By only viewing the last 100 revisions rather than assuring that a set amount of revisions was used for each project chosen, this introduced a threat to validity. The data gathered from a potential 2-revision project is not as reliable as the data gathered from a hypothetical 100-revision project. This is due to having a larger number of revisions to gather data from will reduce the impact of outliers, as well as more correctly generalizing the averages for each data point for each of the approaches.

In order to avoid this threat to validity the revisions for each project should have been run through the criteria until 100 revisions had passed for each project, instead of just observing the 100 most recent project revisions and filtering them through the criteria.

The second criteria of the project selection process could have been done better. It was vaguely defined as "a large number" of revisions.
The purpose of this criteria was to ensure that the number of revisions in the project was high enough that selecting the project for the experiment produced a useful amount of test data, and to make sure that the project was of decent size and updated frequently.

The criteria of “large enough” should have been replaced with a numerical minimum amount of revisions in the project, in order to ensure that the experiment is as scientific as possible.

In addition to this, due to the duality of purpose of the number of revisions criteria, an additional criteria should have been added regarding the frequency at which the project is updated. This new criteria would serve the purpose of not only ensuring that the project is up-to-date, but also that it has a history of receiving frequent updates.

### 6.2 Threats to External Validity

The more two concepts differ in general, the less of their contents are applicable to each other. When trying to replicate or otherwise reuse the results of this study, the criteria for project selection will matter.

If the four criteria for project selection are followed the generalization will not suffer any threats to validity. The more criteria which are missing from other projects where one attempts to utilize the results of this research, the less certain it is that the results translates to such projects.

A project not fulfilling to the criteria of “contains java class files” introduces a small but simple threat to External validity; it is impossible to say that the conclusions of this research stand true for that project, thus introducing a threat to external validity.

A potential External threat introduced by not adhering to the “large number of revisions” criteria would be if a project has a low number of total revisions, as this would reduce the number of total potential revisions used for the experiment. If future research is going to be based on this research then that future research should aim to have plentiful revisions, in order to prevent special case revisions affecting the overall averages of the approaches.

In contrast to this, for the sake of testing rather than research there is no inherent problem in using the regression test selection approaches on such projects, as otherwise they could not be used on newly started projects.
6.3 Threats to Construct Validity

In order to find the potential threats to Construct validity, the variables produced by the experiment need to be examined, keeping in mind how well each of these variable help answer the research questions they are used for.

Looking at the first research question; “How do static regression test selection approaches compare to dynamic regression test selection approaches in regards to number of tests selected?”, as the definition of this variable is simply “the number of tests selected” there is no difference between the theoretical definition and its actual usage in this experiment.

Observing the second research question, “How do static regression test selection approaches compare to dynamic regression test selection approaches in regard to runtime?”, the definition of runtime as used in this experiment is to be analyzed. The runtime measured was defined as the time from the start of the execution of the testing tool up to the time where the execution had been completed entirely and the user could enter the next command into the program which executed the regression test selection approaches.

The intention of this definition for the experiment was to as accurately as possible measure the time during which the user had no input and would have to wait for the program to finish executing. The theoretical definition of runtime is “the time for which a computer software program is executing”, which means that the difference between the theoretical definition and the definition of the runtime variable used in this experiment is small enough to be negligible.

The third research question “How do static regression test selection approaches compare to dynamic regression test selection approaches in regards to safety?” uses the Safety variable. Safety as used in this research is closely related to the classical definition of “recall” used in computer science, which is a measurement of the fraction of relevant items selected from a set of items over the total amount of relevant items in that set.

However as the purpose of this research is to compare the performance of different regression test selection approaches, the measurement of recall is not exactly what is needed. Recall also assumes that the relevant and irrelevant items of the set are known beforehand; this is not the case for this research. Instead what is used in the experiment
is the measurement of Safety, which is a relative comparison of the fraction of relevant items selected by two different solutions. Safety is a variable which is calculated from the amount of tests selected by two different regression test selection approaches.

These two numbers are then put into the formula which calculates the Safety. In order to answer the research question regarding Safety, the numbers produced by the Safety formula are used as the measurement of relative Safety. As the research question is all about Safety as compared between two approaches, using a relative measurement is fine.

The fourth research question “How do static regression test selection approaches compare to dynamic regression test selection approaches in regards to precision?” uses the Precision variable, however the Precision used in this research is different from the classical definition. The classical formal definition of Precision is that it is a measurement of the fraction of relevant items selected over the amount of total items selected.

As this measurement assumes that the relevant items are known, and it only produces the absolute value of precision for one approach it is not of use for this research. However previous research [1] has used a relative Precision measurement to compare the precision performance of two item selection approaches, which does assume knowledge of which items are relevant beforehand. The definition of relative Precision is the one used in this research when referring to Precision, and it is defined in chapter 4.4.1. As this definition is a simple formula which utilizes the items selected by two different approaches, there is not a difference between the theoretical definition of relative Precision and the practical use of Precision in this research.
Chapter 7

Conclusions

Comparing the runtime, Safety, and Precision values of the static regression test selection approaches show that for this experiment, the limited transitivity approach yielded a reduced runtime at the cost of Safety, with a small Precision gain. Introducing a level limitation of the transitivity aspect caused a loss of safety in HKRTS as compared to STARTS.

As the static approaches in general have a safety loss compared to the dynamic approaches this means that the limited transitivity property of the level used in this research is not a solution to the performance problem in static approaches. The safety losses of HKRTS were severe enough that the approach would not be suitable for stand-alone use, which negates the purpose of using a regression test selection approach by forcing the developer to combine the regression test selection tool with running the full test suite.

However, as the run time and Precision of the static approaches were improved by the limited transitivity property, limited transitivity static approaches may be a viable alternative to dynamic regression test selection approaches. If the limit level was to be increased beyond that which was used in this research, it could possibly cause an increase in the runtime while potentially also reducing the loss in Safety. If this “runtime for Safety” trade off could be made without affecting the precision of the limited transitivity approach it could potentially be a viable alternative to the dynamic approaches. This would be the cases where equal runtimes for limited and non-limited transitivity produced lower safety- and precision losses for the limited approach, as compared to dynamic regression test selection approaches.
For this reason I propose that further research within the field is done, in order to increase the potential performance of general static regression test selection approaches. It would be of importance that this future research compares performance of different levels of limited transitivity, because higher levels of limited transitivity property may result in a more safe approach while still retaining the runtime improvement aspect of the property. Level limits close to what was used in this research will most likely result in similar performance changes so any future research will benefit from setting significantly higher level limits.

It would also be of value to try to find an algorithm for a dynamically changing transitivity level for the limited transitivity property. The purpose of such an algorithm would be to find an optimal level limited for each software project it is used on. This algorithm could be based on previous revisions in the project being analyzed as well as data gathered from other projects.
Bibliography


[31] M. Skoglund, P. Runeson, "Improving class firewall regression test selection by removing the class firewall". In International

