Information-Coverage Based Integration Test Selection

Master Thesis Computing Science

de Goede, Tom
tom.degoede@afas.nl

Prasetya, Wishnu
s.w.b.prasetya@uu.nl

Overeem, Michiel
michiel.overeem@afas.nl

August 2023

Abstract

Modern software development approaches promote automated testing as key concepts for quality control. During development software changes are continuously integrated to the mainline codebase requiring test suites to be reviewed. With the growing nature of codebases and the recent monorepo trend the industry has implemented solutions for incremental software building and testing. These solutions work well for unit tests that have a limited set of dependencies but fall short of integration test scenarios where large parts of the codebase are dependent upon. In this thesis we survey theoretical research that uses code coverage data for test case selection. Using these methods we extend the existing model of build systems with coverage concepts and implement this in a modern build system called Bazel. Additionally we propose a generalization of the code coverage algorithm as information coverage that further reduces the need to execute integration tests. Finally we research the efficiency of our methods in a real industry setting and find that we can reduce test case selection rates by 62% using code coverage based selection and 73% using information coverage based selection.

Table of Contents

Chapter 1

Introduction

1.1. Motivation

During the life-cycle of a software system the set of features it must support is likely to grow. Research has shown that a developer should receive feedback about the correctness of his changes as soon as possible (Hans Dockter, 2019). Known as the developer feedback cycle, this is measured as the time between a developer making a change to the codebase and receiving feedback about the correctness of that change. The shorter the feedback cycle is the more productive a developer can be. The ideal length of this feedback cycle is therefore zero seconds. However, in practice, this is not possible, forcing the industry to find a balance between the feedback duration and the amount of correctness it guarantees. The industry has found that a feedback cycle of 10 minutes is acceptable (Bell, 2001). This allows the developer to remain focused on the task at hand and not be fatigued by context switching to other tasks. This allows him to easily recall the context of the change and fix any issues that may arise.

Modern software engineering methodologies such as agile development (Fowler, Highsmith, et al., 2001) and DevOps (Ebert, Gallardo, Hernantes, & Serrano, 2016) promote the use of software defined test scenarios (Meszaros, Smith, & Andrea, 2003) that describe the expected behavior of the software to validate its correctness. Typically these automated test cases are validated against software changes by a continuous integration (CI) system. This system closes the developer feedback cycle by providing test results to the developer. Referred to as regression testing, this process ultimately raises a significant challenge when software changes occur more frequently than a CI system can validate them. Evidently, with a growing set of features that a software system must support, the set of test scenarios also grows and the time required to validate them increases.

Another important insight from the industry is that reducing codebase complexity reduces developer cognitive load which in turn can increase their output (Potvin & Levenberg, 2016). One way the industry is reducing codebase complexity is by using a monorepo (Brousse, 2019; Brito, Terra, & Valente, 2018) instead of a polyrepo codebase. A monorepo is a single repository that contains all the code a company owns or operates. This differs from a polyrepo setup where the code is distributed in many separate repositories. Monorepos remove the need for context switching between different repositories, simplify dependency management and enable better code sharing, improved collaboration and discoverability. Another way the industry is reducing codebase complexity is by employing trunk based development (Henderson, 2017). Trunk based development pushes engineers to always work on the same version of the codebase, the trunk, as the rest of the company. This means that they should keep their changes as small as possible and integrate them as soon as possible. Even though these practices are meant to boost developer productivity they work adversely with regards to the feedback cycle of the CI system. Trunk based development increases the frequency of CI runs whereas a monorepo increases the size of a single CI run.

Modern build systems, such as Bazel (*Bazel*, 2015), have been specifically designed to solve this problem by only building changed parts of a codebase. Their incremental nature decorrelates CI duration with the overall size of the codebase, allowing small software changes to integrate efficiently into a large codebase. The fundamental principle behind these build systems is the representation of the build process as a directed acyclic graph (DAG), comprising interconnected build actions and their dependencies. These actions only require re-execution when their dependencies change. Notably, the dependencies of these actions extend beyond codebase files and can include the output values generated by other actions. Consequently, any code modification efficiently traverses the build graph, ensuring minimal re-execution of build actions, this is referred to as *build minimality* (Mokhov, Mitchell, & Peyton Jones, 2018).

This dependency graph algorithm also works well for unit-tests that are defined on specific parts of the codebase. When defined properly, unit-tests only depend on parts of the codebase that they are actually testing. By only recompiling tested components when they change and subsequently only retesting the compiled binary when compilation resulted in a change, the test duration will depend less on the overall size of the codebase.

Unit-tests, however, come with certain drawbacks. They validate that individual components of the system keep working but they do not guarantee that the components remain compatible or confirm the continued functionality of the final software product. Unit tests can also be time consuming to maintain because they often test internal aspects of the system, which may undergo revisions or reconsiderations at any given time.

A different kind of quality control that circumvents the aforementioned limitations is integration testing. As opposed to unit-tests, integration-tests run after integrating various or all software components into the final product. These tests often simulate real user interactions and describe the expected behavior of the application. They are therefore less likely to change due to internal refactoring and are also easier to define by describing actual user behavior.

A field where integration tests are already being favored over unit tests are low-code platforms (Overeem et al., 2022). These platforms represents a notable industry trend that further elevates the level of abstraction in software systems. The philosophy of low-code platforms is to enable non technical individuals to build software products without programming knowledge. Commonly referred to as citizen developers, they are domain experts in the specific products they are building. As such, they require a functional and thoroughly tested application to meet the unique requirements of their respective fields. However, the technical nature of unit testing, combined with the intricate internal workings of low-code platforms, often renders the creation of unit tests unfeasible or even impossible within these platforms that intentionally conceal their internal complexities.

The drawback of integration-tests, however, is that executing them can be time consuming. Moreover, the incremental graph-based build will have minimal impact on these tests since a big portion of the codebase is part of the test dependencies via the integrated product. Consequently, integration tests tend to run more frequently and require a longer execution time than unit tests. As a result of these factors, the feedback cycle for integration tests is often significantly longer than that of unit tests. Therefore, unit tests currently remain the preferred method of testing in the industry.

To address the benefits of integration testing and the growing demand for them in the context of low-code platforms, we research a possible solution to the limitations of incremental build graphs. Known in the literature as test case selection, this field aims to reduce the number of test scenarios that have to be executed after a software change while still guaranteeing program correctness. A notable test case selection method is Code-Coverage based test selection as introduced by (Rothermel, 1996) and applied by (Beszédes et al., 2012). Code coverage of a test scenario is defined as the code that was executed during the test execution. We believe that the technological advances of build systems now make it more manageable to implement this test selection method in such a way that can benefit the industry for both unit-test and integration-test selection. By leveraging Code-Coverage data, we can significantly reduce the dependencies of a test, limiting them to only the subset of code that was covered during a previous execution. This allows us to reduce the amount of tests that have to be executed after a software change, which improves the minimality of build systems and ultimately accelerate the developer feedback cycle.

Software testing often relies on more than just source code. Test scenarios may involve interacting with databases that already contain pre-existing data, but only where a subset of that data is loaded during test execution. Similar to code coverage, only changes made to this specific subset of the database have the potential to impact test results. We propose a reinterpretation of the Code-Coverage algorithm and introduce a novel approach called the Data-Coverage algorithm. We refer to the combined Code-Coverage and Data-Coverage algorithms as the Information-Coverage algorithm. The Information-Coverage algorithm encompasses both the traditional tracking of code changes and the consideration of changes in the covered subset of data. By integrating these two instances of coverage, we aim to create a more powerful and effective approach for test selection. Furthermore, we explore the possibility of other instances of Information-Coverage beyond Code and Data. We seek to identify additional forms of information that can be leveraged for test selection. This work will contribute to a more comprehensive understanding of Information-Coverage and its potential applications in software building and testing.

1.2. Research Questions

The problem that we are trying to solve is the same Test Case Selection Problem that Rothermel and Harrold introduced. From their work we distil definition 1. In section 5.1. we use this definition in combination with the build system model from section 4.2.1. to formally define how code coverage based test selection can be incorporated into the generic build system model. These model enhancements enable us to implement code coverage based test selection in modern build systems.

Definition 1 (Test Case Selection Problem)
Given: *A program P, modified version of P, P', test suite T and results $T(P) = \{(t, v) \mid t \in T, v = t(P)\}$.*
Problem: *find a subset $T' \subseteq T$ such that any change in $\Delta(T(P), T(P'))$ is also found by $T'(P')$.*
Note: *$T(P)$ represents a set of test cases (t) and their outcomes (v). Here v represents the test verdict: pass or fail. $T(P') \setminus T(P)$ will contain all tests whose outcomes would be different when T was executed on P'. Therefore $T'(P')$ must be a superset of $T(P') \setminus T(P)$.*

To verify that our contributions solve the test case selection problem for integration test scenarios using coverage information we formulate the following research questions:

1. *How effective is Code-Coverage based test selection on integration-test scenarios?*

   To answer this, we will enhance the build system model with coverage data and implement a prototype of this in an existing build system. Using this implementation we will conduct experiments on an existing codebase where we compare test case selection rates against the previous build graph baseline.

2. *Can we generalize to Information-Coverage and does it improve our efficiency?*

   On top of the prototype from question one, we will implement another variation of information coverage in the form of data coverage of a database snapshot. With this implementation we will also conduct experiments, and compare those to the results from question one.

3. *Do our methods yield good enough results for the industry towards changing its testing strategy?*

   By analyzing the results from the previous questions we will get an understanding of how codebase size and changeset size relate to the amount of tests that are being selected. Using this information we can determine the scalability of a test suite of integration tests.

1.3. Contributions

In this thesis we will contribute the following: By extending the existing formal build system model with new concepts we add support for code coverage based test selection. We also contribute a proof of concept that implements these model enhancements in a state of the art build system.

Using this prototype we conduct various experiments to prove that our model enhancements work and measure their benefits. Our experiments are performed on a real world industry system that is in active development. With this method we try to bridge the gap that exists in this field between academia and the industry. Additionally, we generalize the concept of code coverage based selection to information coverage based selection. This new form of coverage tracking allows the encompassing of other types of changes such as data changes into the build analysis. Using this we further reduce test case selection rate. We even find that traditional code coverage based selection is insufficient to measure and detect all changes to a modern codebase. Because of this our generalization to information coverage is required for a sound and complete application of coverage based test selection.

1.4. Structure of this Thesis

The rest of this thesis starts off by giving background information about various core concepts that this work evolves around such as: software testing, code coverage and continuous integration. Chapter 3 follows with related work about different test selection methods and software testing approaches. Chapter 4 goes in depth about build systems and the concept of content hashes and Merkle Trees that our work heavily depends on. Our problem statement is formally presented in chapter 5 where we also describe the research method for each research question. In chapter 6 we extend the build system model from chapter 4 with additional concepts to facilitate coverage based selection. Details about the prototype that we build are given in chapter 7 where we finish with some limitations. Our results are finally presented in chapter 8 and we conclude this work in chapter 9 by answering our research questions. Some ideas for future work are given after that.

Chapter 2

Background

Software testing has been around since the early days of software engineering. C.Baker's review (Baker, 1957) of "Digital computer programming" (McCracken, 1957) is the first to distinguish testing and debugging as two separate phases. He defines software testing as *"the process of making sure that the program solves the problem it is intended to solve"* whereas debugging is *"the process of making sure that the program does what the coder meant to do"*. Contrary to these definitions, in modern day software development, testing solutions are being deployed to catch newly introduced bugs on existing functionality. Referred to as regression testing, this process often involves the execution of a set of automated test scenarios that describe the expected behavior of the software. One of the pioneers in this field was William Elmendorf, who in 1967, while working at IBM labs, formalized a structured approach to software testing in *"Evaluation of the Functional Testing of Control Programs"* (W. Elmendorf, 1967). Not much later, (Dijkstra et al., 1970) is famously quoted for *"Program testing can be used to show the presence of bugs, but never to show their absence!"*.

As software systems continued to expand, their test-suites were also expanding and required to constantly validate new software modifications. Consequently, research began emerging on the topic of test case selection. The goal of this research was to reduce the amount of required tests to validate a software modification to a feasible amount while still guaranteeing program correctness. According to (Yoo & Harman, 2012), one of the earliest documented approaches in this field was by (Fischer, 1977) who formulated the test case selection problem as an Integer Programming optimization function. Recent industry best practices even require every small code modification to be validated for correctness as soon as possible. This process is referred to as continuous integration (CI) which further emphasizes the need for efficient test case selection.

Continuous integration is often performed by so called build systems. These systems are responsible for 'building' the software by compiling source code and packaging the final product. Most of the time these systems are also responsible for executing the test scenarios, only resulting in a successful build when the tests succeed. The industry has developed various build systems such as Make (Feldman, 1979) and Bazel (*Bazel*, 2015). These generic build systems model the build process in a dependency graph which we will describe more in depth later. Thanks to this graph they already perform some form of test case selection by only executing tests that depend on the modified code.

Various test case selection approaches use some form of tracing during test execution to measure the runtime behavior of a test. An important concept here is code coverage, which measures the subset of code that was executed during a test. An early academic mention of code coverage is by Elmendorf (W. R. Elmendorf, 1969), who talks about using branch coverage for testing operating systems. (Piwowarski, Ohba, & Caruso, 1993) describe how code coverage measurements were performed at IBM in the late 1960s. They describe how, initially, a hardware tool was used to measure statement and branch coverage. In the late 1970s, however, a software team proved that a software tool could achieve the same results at a fraction of the costs.

In 1996 Rothermel et al. (Rothermel, 1996; Rothermel & Harrold, 1997) suggested techniques for test case selection using code coverage traces. They found that determining *fault revealing test cases* that were affected by a code change is NP-hard. As evidence they showed that a code modification could cause a program to run forever. Analogous to the halting problem, which is known to be NP-hard, this proved that determining the effects of a code modification to test cases is also NP-hard. Because of this there is no effective procedure that precisely identifies the fault-revealing tests.

Going further, Rothermel et al. determined that in order to be fault-revealing a test case should be modification-revealing but unfortunately this remained NP-hard to solve. They did however find that a test case could only be modification-revealing if and only if it is modification-traversing. This results in the following subset relations:

$$faultrevealing \subseteq modificationrevealing \subseteq modificationtraversing \tag{2.1}$$

To efficiently determine modification-traversing test cases Rothermel and Harrold used code coverage traces of a previous test execution. Namely, given a test case, they tried to find the intersection between the code modification and the code coverage trace. if this turned out to be non-empty, the test case was modification-traversing and had the be selected for retesting. They proved that this heuristic is safe under the *Controlled Regression Testing Assumption* which reads:

Definition 2 (Controlled-Regression-Testing Assumption) *When program P is tested with test t, all factors that might influence the output of P, except for the code in P, are kept constant with respect to their states when P was tested with t.*

Surprisingly however the concept of code coverage based test selection remains scarcely used in the industry. According to a literature review by Ali et al. (Ali et al., 2019) this is due to the terminology differences between industry and academia, making it difficult for practitioners to know what to look for in academic literature. Additionally, many empirical studies are conducted in controlled experimental environments that differ greatly from the complexity of an industrial context. Finally, according to Engström and Runeson (Engström & Runeson, 2010), the existing industry evaluations are also hard to compare due to the many differences in context.

Chapter 3

Related Work

In a recent publication Ivanković et al. from Google, (Ivanković, Petrović, Just, & Fraser, 2019) describe their coverage collection system that operates on a immense industry scale. Built upon their internal build system Blaze (Henderson, 2017), which is open sourced as Bazel (*Bazel*, 2015), this pluggable system is capable of collecting coverage information from any programming language (of which they have implemented seven). The collected line coverage information is then used to give developers insights during their workflows of authoring changelists (code modifications) and reviewing such changelists. Interestingly, the authors state that coverage collection is only enabled for unit-tests since line coverage is not the best suited coverage measure for integration scenarios. Another thing to note about their publication is the issues they encounter when enabling coverage collection. Mainly due to additional overhead imposed by instrumentation, the tests are more likely to timeout or surface other kinds of issues. Therefore they only execute code-coverage after executing the unit-tests normally. In their work Ivanković et al. never mention using the coverage information for test case selection.

3.1. Test Selection Approaches

The 2012 survey by Yoo et al. (Yoo & Harman, 2012) recognized 12 different test case selection approaches ranging from Control and Data Flow Graph walk algorithms to Symbolic Execution. While certain techniques only consider the static information available, most methods use some form of tracing during execution of the baseline program. After applying a change to the baseline program the selection algorithm will find the intersection between the baseline traces and the changeset. When this intersection is non-empty a test is labeled as *modification traversing* and has to be selected again. The work of Yoo et al. categorized the different testing techniques into three distinct categories: test case minimization, which tries to find the minimal set of tests to cover a specific program modification, test case prioritization, which tries to prioritize certain test scenarios over others by for example running fast failing tests first, and our most relevant category, test case selection, which tries to find all relevant tests for a program modification ultimately being the only safe category.

Another survey by Mukherjee et al. (Mukherjee & Patnaik, 2021) focuses on test case prioritization. They studied 90 different papers on the subject and found many code coverage based techniques. They however also conclude that collecting code coverage is tedious and costly in many cases. This emphasises the importance of our research. A publication by Elbaum et al. (Elbaum, Rothermel, & Penix, 2014) combines test case selection and prioritization but also explicitly states that they do not rely on coverage information. Their reasoning here is that code coverage information quickly becomes inaccurate when a codebase changes frequently (Elbaum, Gable, & Rothermel, 2001).

A radically different approach was researched by Elsner et al. (Elsner, Hauer, Pretschner, & Reimer, 2021). They applied machine learning (ML) and investigated to what extent Version Control System (VCS) data, such as authors and changed files, and historical CI information, such as failed tests, would suffice for test case selection. With different unsafe approximation approaches they managed to achieve 90% accuracy with a time savings of 84%. Their reasoning for only looking at these data sources was that this information is readily available in current industry deployed pipelines (as opposed to code coverage data). Interestingly, they also found out that traditional heuristic based approaches often outperform complex machine learning models.

Even though static analysis methods such as symbolic execution can theoretically reduce the set of dependencies of a test case they are infeasible to deploy at scale on integration-tests. Additionally, modern build systems allow for more aggressive dependency pruning after executing a test by collecting runtime data, such as code coverage, to do so. Since we are looking for a safe algorithm, that finds every tests for which the verdict may have changed, we narrow down the set of test case selection methods to those that collect runtime coverage information such as traces or hit locations.

Although path coverage methods as described by Schalkwijk et al. (Schalkwijk, 2011) are theoretically more precise than the Rothermel-Harrold algorithm, Schalkwijk concludes that this is actually very synthetic and unlikely to occur in real world applications. The added overhead of tracking paths instead of hit locations does not outweight this theoretical benefit.

3.2. Live Unit Testing

Live unit-testing, or continuous testing solutions (Joe Morris, 2017) have faced the same challenge of test suites becoming too large with respect to the change frequency. The goal of these solutions is to display up-to-date test results while a programmer is working in an Integrated Development Environment (IDE). Especially useful when applying Test Driven Development (TDD), they share our goal of reducing the feedback duration for an engineer to increase productivity. Not so surprisingly they also explored the use of code coverage information for test selection. Interestingly, Greg Young, the author of Mighty Moose (Young, 2012), states that code coverage is not a good solution for the problem as it describes the last run, which may have nothing to do with the current run. We believe that the controlled regression testing assumption from Rothermel et al. (definition 2) resolves this problem and can be applied to software testing.

3.3. Potential Applications

The generic dependency coverage information that is produced and stored in the enhanced build system model (see 7.8.3.) could also be applied to various other testing practices. We explore some of these in this section.

3.3.1. Similarity based test prioritization

As described in Neto et al. (de Oliveira Neto, Ahmad, Leifler, Sandahl, & Enoiu, 2018), similarity based test selection tries to find a minimal set of tests that cover the set of code changes. It achieves this by removing test cases that only cover code that was already covered by other tests, ultimately reducing redundancy in test scenarios. This is an efficient method to speed up the feedback process for the software engineer because it makes sure that tests that likely have the same result as other tests are dropped. We should note that this method is not safe because tests with overlapping coverage may assert different aspects of the program. Yoo et al. referred to this concept as test case minimization. Our model enhancements add generic code coverage data per test to the build system. This information can also be used to implement similarity based test selection.

A challenge would be to adapt the build system to support it since build systems are generally designed to (incrementally) build everything that may produce different outputs, rather than a subset. Similarity based test selection can also be interpreted as a test case prioritization algorithm. Where traditional testing would sequentially test each program component, using similarity based scheduling would cover all parts of a code modification before proceeding with other test scenarios. This has the same benefit of speeding up the feedback cycle but also remains safe since other tests are still executed later.

3.3.2. Mutation testing

Another interesting topic of research is mutation testing (Petrović & Ivanković, 2018; Petrovic, Ivankovic, Kurtz, Ammann, & Just, 2018). Mutation testing takes a program and generatively inserts modifications in the program that are meant to break it. This gives a very good indication about the quality of a test suite by validating how good it is in detecting these faults. Often this is considered a better indicator of test suite quality than code coverage since covering a line does not imply that it is being asserted. The downsides of mutation testing however are that there are often a lot of mutations to consider, it is unclear which mutations are applicable for a change and which tests should be executed to traverse a mutant (Ojdanić, 2022). Using the coverage information that is available in our new model the set of mutants can be reduced to the change affecting mutants that only have to validate against change traversing test cases. Next to code coverage this could be a valuable indicator of the test suite health with regards to a contributed code change.

Chapter 4

Preliminaries

In this chapter we will describe the formalities around a build system. We require these definitions to derive our own enhanced model that includes the possibility to perform code coverage based test selection. We will first introduce the concept of hashing because we heavily depend on it.

4.1. Hashing & Content Addressable Systems

Various technologies use hashing algorithms to compute a compact representation of a piece of data. Referred to as a hash or digest, the output values of these algorithms are designed to be a unique, fixed length sequence of bytes for each piece of arbitrary input data. Hashes can then be used to validate the integrity of a piece of data, by recomputing its hash, to check of a piece of data has changed or as a cache key to store the data in a so called Content Addressable Storage (CAS).

4.1.1. Merkle Trees

More complex datastructures that are comprised of hashes exist. A notable variant is a Merkle Tree (Merkle, 1979) which, instead of representing a single piece of data, can represent a tree of data. Merkle Trees achieve this by representing each node in the tree as a list of its child trees and leaf nodes, identified by their hashes. Stored under their hash in a CAS, they allows to recursively resolve any child nodes of a given root Merkle Tree node. A folder structure could, for example, be represented as this:

```
root/
  index.txt
  a/
    file1.txt
    file2.txt
  b/
    file.zip

// This structure is then represented as the following three merkle tree nodes, which
    themselves are stored under their own hash in a CAS

// root
hash(index.txt) index.txt
hash(a) a
hash(b) b

// a
hash(file1.txt) file1.txt
hash(file2.txt) file2.txt

// b
hash(file.zip) file.zip
```

A widely adopted tool that extensively uses hashing is the Git version control system (VCS). It stores each revision of a software system in a Merlke Tree format which allows to reuse most Merkle Tree nodes between revisions. This results in a compact representation and these different revisions are also very fast in comparing and determining the changed set of files. Only Merkle Tree nodes for which the hash has changed must be recursively considered.

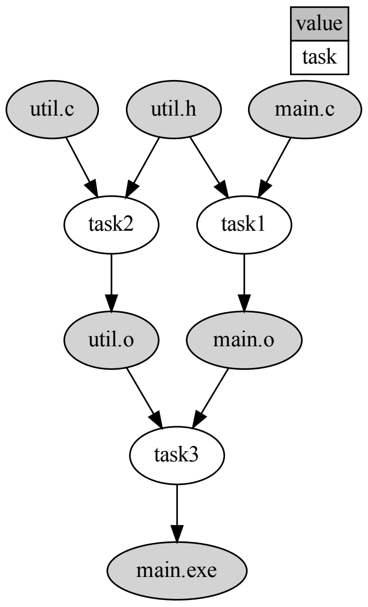
4.2. Build Systems

Build systems à la Carte (Mokhov et al., 2018) formalizes the core of build systems and dependency graphs. They describe build systems as tools that automate the execution of repeatable tasks at scale from individual users to large organizations. In their paper they classify and categorize four different build systems by distilling a set of common properties and their different approaches. Surprisingly they also describe Excel as a build system in disguise. They argue that each individual cell is a target that can depended upon by other cells. The authors describe an abstract model to describe any build system and provide a concrete implementation of this model in Haskell. For clarity we will repeat the necessary parts of the model here, and later expand on this model with our own contributions.

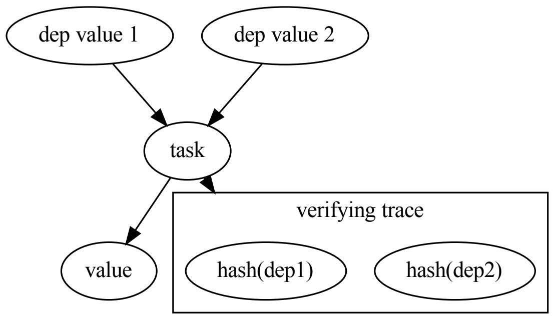
4.2.1. Build System Model

At the core of any build system there are **keys** and **values**. The goal of the system is to respond to build requests by bringing the corresponding values of requested keys up to date. In software build systems these keys and values represent the filesystem. A build request could for example try to build *main.exe*. The build system is then expected to provide a value by writing the *main.exe* file to the filesystem.

To achieve this, build systems model the build execution as a set of **tasks**. These tasks describe how to compute one or multiple values for a given key. Depending on the type of build system these descriptions could already contain a static list of **dependencies** to other values. Other systems allow dynamic dependencies to be resolved during the execution of a task. Another way that we refer to task dependencies is as **input values**. Because input values of a task are possibly computed as output values of other tasks this system forms a Directed Acyclic Graph (DAG) of tasks connected by values. When a task initially defines a static set of dependencies but dynamically reduces the size of this set during execution, this is referred to as *dependency graph pruning*. An example of a dependency graph for program main.exe is given below.



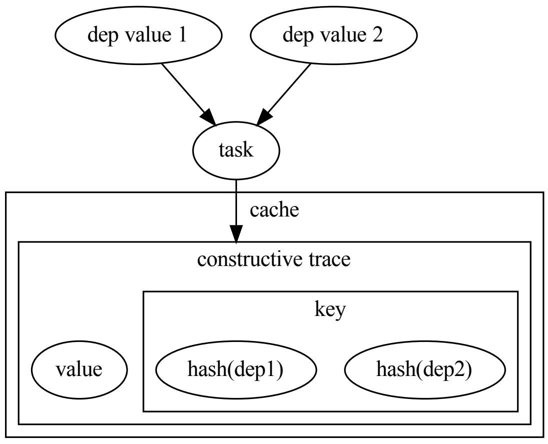
To be more efficient, build systems often perform up to date checks on values. This allows them to skip the execution of a task, when it is sure that the value will not change, resulting in **incremental builds** (Maudoux & Mens, 2018). Most often this is done by first bringing the present set of dependencies up to date and then validating that the existing value was computed with this same set of dependencies. In other words, none of the dependencies changed since last computing the value. Most build systems implement this by tracing modification timestamps or content hashes of dependencies when a task is executed. The paper refers to these traces as **verifying traces** because they are used to verify that the existing dependency values have not changed since a previous execution of the task. Verifying traces therefore are additional metadata that a build system produces to improve its efficiency. A schematic example of such trace is given below.



The benefit of using content hashes over modification timestamps is that they are possibly less volatile. When a task is retrigged due to dependency changes it may very well produce the exact same result. A requirement for this to function properly is that the task produces their values deterministically. When a build system only executes tasks when their dependencies have changed the build system is minimal. (Mokhov et al., 2018) define the notion of build system minimality as:

Definition 3 (Minimality of a Build System) *A build system is minimal if it executes tasks at most once per build, and only if they transitively depend on inputs that changed since the previous build.*

Another way build systems are speeding up builds is by caching output values of tasks in a separate cache. **Caching builds** (Maudoux & Mens, 2018) are achieved by extending on the model of verifying traces and adding the actual output values to the traces of a task execution. The cache then allows for lookups of task output values given the hashes of the task dependencies. This cache is often keyed by another hash of all the combined task dependency hashes. Referred to as **constructive traces**, because they facilitate reconstructing the value, these traces can be shared between multiple build invocations at different moments in time. This cache allows to easily switch between different versions of a codebase without having to execute otherwise invalidated tasks again. When this cache is stored in a shared network location it facilitates distributed builds and sharing build results between many developers. This is often referred to as a **remote cache**. Again we give a schematic representation of a constructive trace below.



When regarding the execution of a test scenario as a build system task, and the test result as a build system value, we can use the previously described concepts of up-to-date checks, dependency graph pruning and caching to reduce the amount of tests that need to be selected after a software change.

Chapter 5

Research Method

5.1. Problem Statement

Let us recall the definitions for the Test Case Selection Problem and Minimality of a Build System.

Definition 1 (Test Case Selection Problem)
Given: *A program P, modified version of P, P', test suite T and results $T(P) = \{(t, v) \mid t \in T, v = t(P)\}$.*
Problem: *find a subset $T' \subseteq T$ such that any change in $\Delta(T(P), T(P'))$ is also found by $T'(P')$.*
Note: *$T(P)$ represents a set of test cases (t) and their outcomes (v). $T(P') \setminus T(P)$ will contain all tests whose outcomes would be different when T was executed on P'. Hence $T'(P') \supseteq T(P') \setminus T(P)$.*

Definition 3 (Minimality of a Build System) *A build system is minimal if it executes tasks at most once per build, and only if they transitively depend on inputs that changed since the previous build.*

As introduced by Rothermel and Harrold, the Test Case Selection Problem from definition 1 requires that testing P' with T' will execute at least all test cases in T of which the outcomes have changed since $T(P)$. When interpreting the execution of a test scenario as a build system task we can read the Test Case Selection Problem as trying to optimize our minimality function from definition 3 for a test task. It is worthwhile to point out that the output value of a test task represents a verdict if the task succeeds or fails. The test task itself is not responsible for building program P, it only verifies test t against program P. Because this process may be computationally expensive we seek ways to improve minimality. Given the definition of Minimality, the subset of required test tasks can be reduced by reducing the set of inputs to these tasks. We formalize this in the following definitions:

Definition 4 (Build Task Minimality)
Given: *a task, t, its inputs I and its output o*
Problem: *Find a subset of I, I', used by t to compute o such that $t(I) = o = t(I')$*

When a build task verifies a test scenario, t, on P we consider program P an input of test task t. The test selection problem can then be answered for each individual test case as:

Definition 5 (Test Task Minimality By Program Coverage)
Given: *a test task t, a program P, its modification $\Delta(P, P')$, test task inputs $P \subset I_t$ and a used subset of inputs, $I'_t \subseteq I_t$.*
Problem: *find if any element of I'_t is modified in $\Delta(P, P')$.*

When P itself is an atomic element of I any modification of P causes all test tasks that use P to be selected. We can however think of ways to subdivide P into a set of program parts. This in turn provides to possibility to only include a subset of P in I'.

In the case of code-coverage based test selection we subdivide P by the granularity of our measured coverage. The covered subset of P for test t is defined as $cov_t(P) \subseteq P$, which contributes to the used set of inputs $cov_t(P) \subset I'_t$ instead of P itself. This allows us to more precisely answer definition 5 by finding if any covered element of P was modified: $cov_t(P) \cap \Delta(P, P') \neq \emptyset$. By definition we then select all *modification traversing* test cases. Adhering to definition 1 this is proven safe by Rothermel and Harrold.

5.1.1. Information Coverage

In addition to P, the test inputs I could contain other types of information that are required to compute the test result. An example of this is a database that is used as the baseline database for a test execution. By substituting P in the definition above for anything else we can generalize our selection approach from Code Coverage to Information Coverage. This allows us to subdivide not only a program into many coverable parts but any element of I, such as a database. Definition 5 can then be rephrased as:

Definition 6 (Test Task Minimality By Information Coverage)
Given: *a test task t, an information dependency D, its modification $\Delta(D, D')$, test task inputs $D \subset I_t$ and a used subset of inputs, $I'_t \subseteq I_t$.*
Problem: *find if any element of I'_t is modified in $\Delta(D, D')$.*

Also in the example of a database we could think of D as a subdividable set of pieces of information instead of being an atomic element of I. This in turn allows us to measure which pieces of data were actually loaded during the test scenario. We see this idea of Data Coverage as one more instantiation of Information Coverage.

5.2. Addressing the Research Questions

5.2.1. Question 1

How effective is Code-Coverage based test selection on integration-test scenarios?

We will answer question 1 by implementing Code-Coverage based test selection in the Bazel build system as previously described and use it to experiment on an existing codebase. To enhance Bazel with coverage data we will adopt and adapt Coverlet for C#. This existing library allows us efficiently and incrementally instrument backend code artifacts, enabling them to produce coverage data. Bazel already provides ways to prune the dependency graph of a build task after it was executed. We will add support for this to test tasks and also add the ability to prune portions of a dependency by breaking them up into multiple smaller dependencies. Using the coverage data of the instrumented files enables us to only keep the covered code as final inputs to a test task. This configures Bazel to more aggressively cache the test results. Differently put, this improves test case selection efficiency. As a baseline for our measurements we take the test case selection rate without pruning any code lines from the test task inputs, solely depending on the build dependency graph analysis. By comparing this with the rate that we achieve after applying our method we measure how effective it is.

By answering question 1 we hope to prove industry wide applicability of the previous test case selection research using modern build systems. When proven applicable our research will also open up future research possibilities on software-engineering-best-practices that increase efficiency of the coverage based test selection algorithm. As a contribution to this field we plan to provide a list of challenges we encountered and how we overcame them.

5.2.2. Question 2

Can we generalize to Information-Coverage and does it improve our efficiency?

To answer question 2 we will implement our new concept of Data-Coverage for the same codebase and tests that were used in question 1. Instead of seeding the database during the test scenarios we will seed a snapshot of a database in another build task. By using this database snapshot as a test task dependency we potentially reduce the coverage of the test task itself by removing any coverage that only occurred during the seed process. When the database snapshot is stable this will improve our test selection rate. We will improve further upon this by generalizing our dependency pruning logic to not only prune source files but any dependencies of a test scenario. During test execution we then also keep track of which parts of the database snapshot are actually being read and only maintain those as test dependencies. Similarly, our model driven application consists of a (very) large set of generated JSON definitions. For these we apply the same method, only including them when they are read during the scenario. We will measure the difference in test case selection rate with and without these extensions to answer our second research question.

By answering question 2 we will formalize the concept of Information-Coverage and how this can be expressed in modern build systems. We will experiment with our method on the private codebase repositories of AFAS Software.

5.2.3. Question 3

Do our methods yield good enough results for the industry towards changing its testing strategy?

By analyzing the results of question one and two, for which we performed experiments on a software system currently in development, we determine how likely it is that coverage based test selection scales to larger integration test sets. A crucial aspect here is that for most code modifications the amount of selected test remains small. It is inevitable that some changes will impact all or a very large subset of tests, but as long as this is occasional this is acceptable and can efficiently be incorporated in the developer workflow by scheduling them properly. Using the results of question 1 and 2 we will determine how often this occurs.

By answering question 3 we hope that our results prove useful and that software engineering can benefit from a paradigm shift where integration tests can be validated as efficiently as unit-tests. This would allow test suites to be defined on a higher level by less technical people and to more closely resemble the true behavior of the software end user.

Chapter 6

Extended Build System Model

When projecting definition 6 to the build system model from section 4.2.1., we should note that program P is represented as one or multiple build system **values**. To facilitate our requirements of coverage based selection we extend the model by introducing two new concepts.

6.1. Value Structure Traces

A task can describe the internal structure of its output values in an arbitrary data structure. This data structure subdivides the value into individual pieces of information for which change can be tracked independently. We refer to this metadata as **value structure traces**. To give a better understanding of how they may look, we now demonstrate some variations.

A straight forward data structure to represent the subdivision of information that a value provides is a **map** of information keys and their checksums. This facilitates the tracking of change for each individual key by validating if its checksum changed. For a program consisting of three individual parts the value structure trace could look like:

```
Part1 = hash(Content of Part1)
Part2 = hash(Content of Part2)
Part3 = hash(Content of Part3)
```

An extension to the map data structure is the representention of values as **Merkle Trees** from section 4.1.1. This allows us to describe hierarchical structures of values such as files and folders in zip archives, or classes and methods in source code. This in turn enables a task to measure coverage on different levels of granularity which we will discuss more in depth in section 6.4. Merkle Trees also allow us to model complex dependencies that we will talk about in section 7.8.3. For a program that contains one class with some methods this could look like:

```
Program        hash(Sub Trees of Program)
    Class1     hash(Sub Trees of Class1)
        Method1 hash(Content of Method1)
        Method2 hash(Content of Method2)
```

When the only question that we ask our value structure trace is: "did the checksum for a given key change?", we can simplify the data structure one step further. The same query can be answered by combining the key and its checksum into a single hash value. This allows us to reduce the complexity of the trace to a **hashset**. This basically means that we can track if all covered elements of the value are still present. Unfortunately, when talking about complex dependencies in section 7.8.3., hashsets only suffice under very strict conditions. Adapting the first example that we gave for a map then becomes the follwing hashset:

```
hash(Part1 + Content of Part1)
hash(Part2 + Content of Part2)
hash(Part3 + Content of Part3)
```

6.2. Dependency Coverage Traces

A task can describe the coverage for each of its dependency values. This coverage denotes the subset of pieces of information from the dependency value that were actually used during task execution. This implies that other parts of the dependency value will not affect the output values of the task. We refer to this metadata as **dependency coverage traces**. Again, we introduce some variants to give a better understanding of how this may look.

The simplest structure for the coverage traces is a **map** of covered information keys and their checksum. These keys can then be used to validate if they were unchanged in the dependency value by comparing checksums. This also works when validating against merkle tree value structure traces by substituting keys for merkle tree paths. A dependency coverage trace for a task that only covers part one and three of a dependency value would then look like this:

```
Part1 = hash(Content of Part1)
Part3 = hash(Content of Part3)
```

When modelling value structure traces as a **hashset** it makes sense to do the same for dependency coverage traces. Each traced hash represents a piece of information from the dependency value that was covered. The up to date check simply checks if all hashes in the coverage trace are still present in the value structure trace. This can be implemented as a is-subset-of operation which could result in a more efficient implementation. An adaption to the previous example would then look like this:

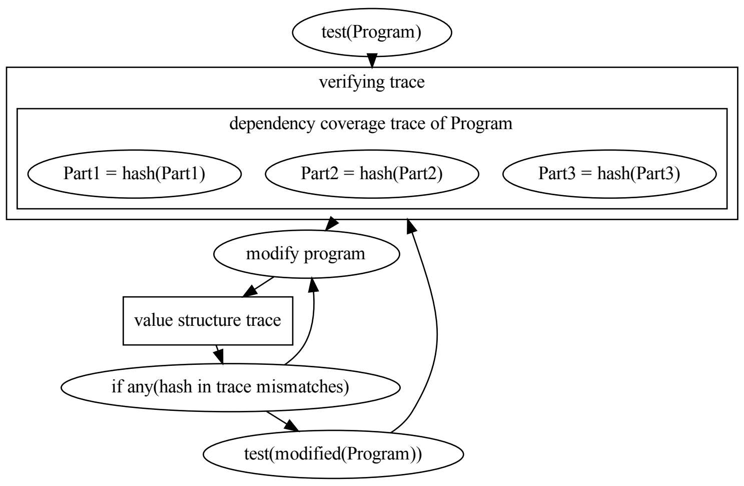
```
hash(Part1 + Content of Part1)
hash(Part3 + Content of Part3)
```

An optimization to both previous variants of a coverage trace could be a list of booleans denoting if a piece of information at their corresponding index was covered. These so called **hit locations** refer to an element of the value structure trace. There are multiple factors that influence when this is more efficient than tracking individual hashes. Especially when the coverage of values is relatively large the resulting coverage trace will likely be smaller when using a boolean list of hit locations. When the program that we used in the previous examples consists of four parts in total this dependency coverage trace would be:

```
true
false
true
false
```

6.3. Integration

Value structure traces allow us to subdivide values, such as program P, into multiple parts. Previously we could only achieve this by splitting actual file values into multiple files which is not desirable in many cases. Dependency coverage traces allow us to reduce the amount of task dependencies, such as test inputs I, by tracking their actual coverage. These additional traces then allow us to improve the incrementality of builds by extending on **verifying traces** and implementing a more precise up to date check. Namely, we can verify if all the covered pieces of information of a dependency value still pass the up to date check. Because this is a subset of the entire value, this is more likely to occur than passing the up to date check for the entire value. The up to date check can be implemented by comparing a previous *dependency coverage trace* to the current *value structure trace* of the dependency value. We illustrate this process in the schema below.



There are some caveats to note when applying this to **constructive traces**, for example in a distributed setting. Because a dependency coverage trace forms a subset relation to a value structure trace it becomes less trivial to find a matching constructive trace in a cache. Previously there had to exist a direct match between all the dependency value checksums and a constructive trace to reconstruct a value from the cache. Because of this direct match we could reduce the cache key itself to a hash enabling very fast lookups. Because of the subset relation there can theoretically be various cached values where all dependency coverage traces form a subset relation to the current value structure traces. Finding these traces in an arbitrarily large cache quickly becomes too expensive.

Imagine a program under test that produces a value structure trace containing three hashes, a, b and c. Suppose the cache contains the following constructive trace keys:

```
a + b
a + d
c + e
```

Ideally we would select the $a + b$ constructive trace since it is compatible with the current value structure trace of a, b, c. In a small example this seems doable since we can check the subset relation for each constructive trace in the cache. When we however imagine that a program consists of thousands of parts, and the cache contains thousands of constructive traces, this algorithm no longer performs.

Current build system implementations therefore uses the unpruned list of dependencies as the constructive traces cache key. In our example the cache key then becomes $a + b + c$. Because of this we can only reconstruct values when all dependencies precisely match. Due to the incremental nature of software changes this however is a lesser problem. We can simply find the latest cache hit by backtracking codebase revisions. This will result in the trace that most likely matches the subset relations because it is closest in the codebase history. After finding a matching trace and reapplying changes again the previously described enhanced up to date logic can be applied. This process of making sure that a baseline trace is present is referred to as warming up the local cache. Currently there is no build system that performs this automatically. Therefore it can be worthwhile to perform this manually in a CI setup for builds where dependencies are aggressively pruned, such as with code coverage based test selection.

6.4. Trace Granularity

An important decision to make is how to subdivide values into pieces of information. This decision influences the granularity of our coverage measurements and therefore the precision of our test selection algorithm. One crucial aspect to consider here is how each piece of information is identified. When none of the covered pieces of information changes between codebase modifications we must be able to identify these pieces of information under the same key as before. A synonym we use for keys when we are talking about code segments is **labels**. We could for example choose to divide a codebase per class or per method. Because classes and methods are named by the programmer they are easily keyed accordingly in the value structure trace. Code coverage is however often measured on a more fine grained level allowing for more precise test selection. Only when the code in a covered branch changes must we revalidate the test scenario. Coming up with a deterministic key for a branch is however less trivial.

Rothermel et al. use a graph walk algorithm to compare the control flow graph of program P and its updated version P'. As long as the covered subset of the control flow of P is unchanged in P' a test is not selected, but when a single node diverges the graph walk fails. We can leverage this to identify intraprocedural constructs such as branches by using the control flow path to a statement as its label. This is true because if the path that leads to a covered branch contains changes, this path must be covered as well, hence the test scenario is inevitably selected for testing. In this particular case we therefore do not care if the branch itself also changed and thus also do not care if the label for the branch changes. Whenever multiple control flow paths converge again, they are collapsed into a single node in the label. This algorithm allows us to implement semi-deterministic branch labeling as demonstrated in the comments of the following example.

```
function m()
{
   a; b; c;

   // m/{a;b;c;if(a)/then}
   if(a) { }
   // m/{a;b;c;if(a)/else}
   else { }

   // m/{a;b;c;if(a);if(b)/then}
   if(b) { }

   d;

   // m/{a;b;c;if(a);if(b);d;if(d)/then}
   if(d) { }
}
```

We should note that these labels can become of arbitrary size when the control flow graph grows. To solve this we simply store a hash of these sequences as labels instead. As far as we are aware this content based labeling method has not been documented before but does feel natural in a content addressable system and strongly resembles the graph walk logic from previous work. The coverage framework LCOV uses GCC branch ids to identify branches but these are unstable after changes. Ivanković et al. (Ivanković et al., 2019) mention that they use a slightly modified version at Google of GCC ids but they do not elaborate on its characteristics or reasoning.

Chapter 7

Implementation

To validate our model enhancements we have implemented a prototype that supports information coverage based test selection. In this prototype we implement code coverage based test selection for C# and data coverage based test selection for a custom database snapshot format in JSON. We do not implement the previously described solutions for missing information dependencies. Therefore our implementation is not yet complete.

7.1. Bazel

We use the Bazel (*Bazel*, 2015) build system to implement and validate our work. Bazel already has support for incremental building and testing and supports integrating custom logic in so called rulesets for new languages or additional requirements. This allows us to easily plug in the coverage collection processes for the different languages that we require. Bazel also implements various concepts that we utilize for caching. This caching behavior translates to answering the test case selection problem because cached tests are not selected.

7.2. Collecting Coverage

Collecting coverage data requires instrumenting the source code. Source code instrumentation is the process of mutating the source code with additional statements that trace the runtime behavior of the code. In the previous chapter we referred to these traces as the *dependency coverage traces*. We use a slightly adapted version of the cross-platform coverage toolkit called Coverlet (Solarin-Sodara, n.d.) for .NET. Coverlet takes as input the DLL containing already compiled C# code, and outputs a similar DLL with the added trace statements. We adapt this process to fit nicely in the Bazel ecosystem, allowing us to cache instrumented DLL files. During a test run Coverlet keeps track of different language constructs such as lines and branches by inserting very lightweight interlocked increment statements on a one dimensional array of integers. After a test scenario is executed this so called hits array is persisted as part of the test results. For our purposes it suffices to only store a single boolean bit per hit location.

We also adapt Coverlet to write a file containing a label for each instrumented hit location during instrumentation. Using two separate files allows us to cache the labels once per instrumented file, and store the boolean array of hits as a bit array per executed test. This data layout is similar to the two-sequence encoding as talked about in Code Coverage at Google (Ivanković et al., 2019). Besides the labels we also track the content hash for each hit location. This ultimately allows us to detect changes to covered code. A noteworthy aspect for our implementation is that these content hashes are based on the IL code instead of the source code.

7.3. Labeling Source Code

While Code Coverage at Google (Ivanković et al., 2019) describes labeling and coverage results on the resolution of line numbers, this will not suffice for our application since the line number for a given program statement may change between P and P'. After obtaining hit locations for an arbitrary test scenario it is important that we can track these locations before and after a change $\Delta(P, P')$ has occurred using deterministic labels. This is required because we want to known if modified code segment $s \in P'$ actually is part of the hits array for test execution $t(P)$. This labeling is trivial when measuring coverage on the granularity of methods and functions since they are uniquely named by the programmer. A small example is given below to illustrate this. It becomes more challenging when we wish to measure coverage on a more fine-grained level that also distinguishes intraprocedural control flow such as branch coverage.

```
namespace Runtime;
class Program
{
    public void Main() => Write(Read());
    public int Read() => 1;
    public void Write(int v) =>
        Console.WriteLine(v);
    public void Uncovered() => Write(2);
}
```

```
Runtime.Program.Main()
Runtime.Program.Read()
Runtime.Program.Write(int)
Runtime.Program.Uncovered()
```

Labels for the Program.dll on the left

7.4. Merkle Tree Artifacts

To make Bazel understand partial changes of our program we implement a way to provide a value structure trace of a file that replaces the default content based hash. This feature allows us to subdivide the program into many parts for which changes can be tracked separately. Bazel represents output files as Artifacts and already knows the concept of Tree Artifacts that represent folders or archives in a Merkle Tree structure (Merkle, 1979). Similarly, we will implement a special Artifact type that uses a provided Merkle Tree based on the virtual tree structure of the file it represents. Existing dependency graph pruning logic can then be adapted to also prune subsets of these special Artifacts similar to the existing Tree Artifact pruning. This results in a reduced set of test inputs.

In our particular case we use the previously described labeling logic to name the Merkle Tree entries and hash the relevant source code content to detect changes between software modifications. This content hash is then persisted during instrumentation right next to the label. Depending on the granularity this can be the entire method body, or just a single statement. The previous list of labels for Program.dll is is then described by the Merkle Tree below. As an illustration we also added a class label as the root level of the Merkle Tree.

```
Runtime.Program              hash(Nested Merkle Tree Nodes)
  Runtime.Program.Main()       hash(Code in Main)
  Runtime.Program.Read()       hash(Code in Read)
  Runtime.Program.Write(int)   hash(Code in Write)
  Runtime.Program.Uncovered()  hash(Code in Uncovered)
```

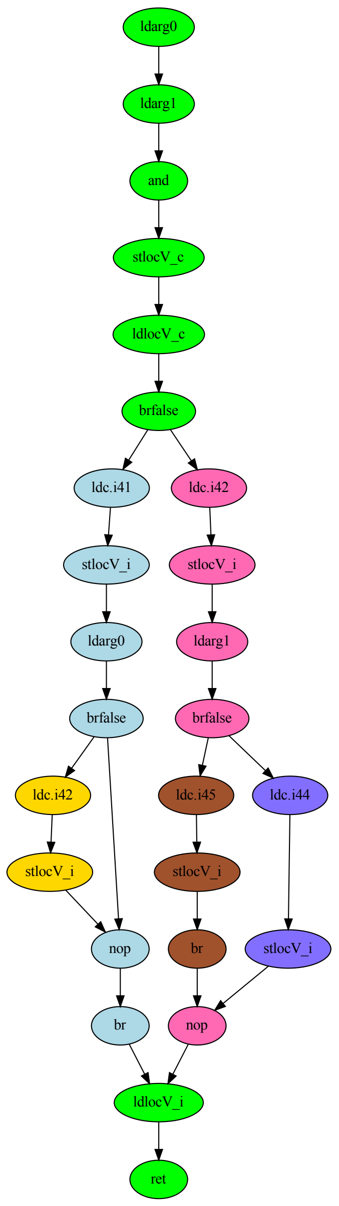
During test case selection we can finally use this information to determine for any covered code segment label of $s \in P$ if their content hash is the same in P' rendering testing unnecessary.

Interestingly, determining if a hash is unchanged in a keyed lookup can be simplified to a set contains algorithm by moving the key inside the hash as shown below. Intuitively this translates to finding if all covered program segments $s \in P$ are still present in P' which is what we also derived in definition 5.

```
hash(Runtime.Program           + Nested Merkle Tree Nodes)
hash(Runtime.Program.Main()    + Code in Main)
hash(Runtime.Program.Read()    + Code in Read)
hash(Runtime.Program.Write(int)  + Code in Write)
hash(Runtime.Program.Uncovered() + Code in Uncovered)
```

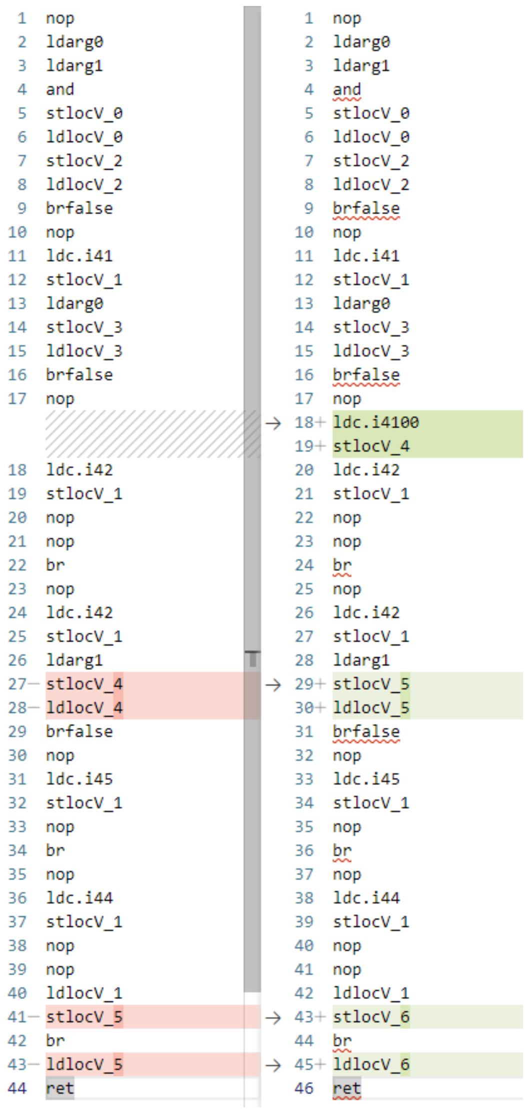
7.5. Branch Coverage

```
int Test(bool a, bool b)
{
    var c = a && b;
    int i;

    if(c)
    {
        i = 1;

        if(a)
        {
            // var y = 100;

            i = 2;
        }
    }
    else
    {
        i = 2;

        if(b)
        {
            i = 5;
        }
        else
        {
            i = 4;
        }
    }

    return i;
}
```

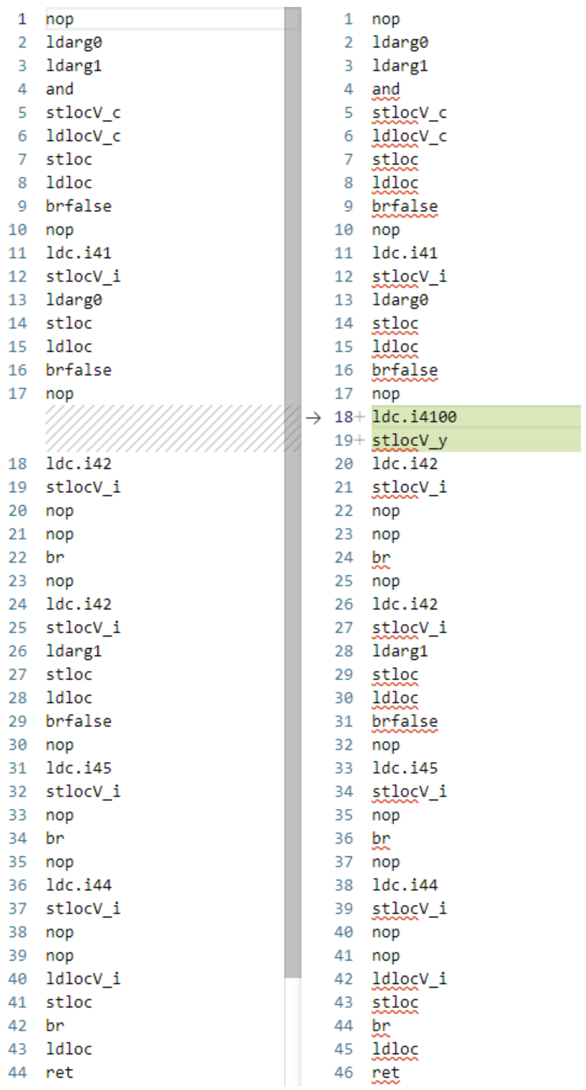


We have already described a theoretical model to tackle intraprocedural change detection in section 6.4. For our C# implementation we however use the the intermediate language (IL) code that is part of the DLL that we are instrumenting to determine content hashes of methods. By reconstructing the IL statements of a method as a graph, we were able to still leverage the graph walk approach of Rothermel and Harrold. An example of a C# function body and IL instruction graph are given above. To trim down graph size we omitted some obsolete statements from it. Using this graph we determined keys and content hashes for C# code blocks and branches. Using IL actually brought some benefits over using the actual source code. Many C# ways of branching were reduced to only a few different IL statements. We also found that symbol resolution, which in IL has already happened during compilation, actually is a crucial requirement for code coverage based test selection. We will discuss this further in section 7.8.1.

One substantial challenge we encountered was that introducing new variables in the method would shift all subsequent variable locations by one. As demonstrated in the example below, uncommenting the y variable actually also caused modifications of IL in branches that were left untouched. The solution we applied here was to replace the IL variable indices with the names the programmer gave them. This still left some unnamed variables that happened to be inserted by the compiler for debugging purposes. These pairs of stclocV_X and ldlocV_X could be stripped of their index entirely because they would always refer to the same index.



```
 nop                    nop
 ldarg0                 ldarg0
 ldarg1                 ldarg1
 and                    and
 stlocV_0               stlocV_0
 ldlocV_0               ldlocV_0
 stlocV_2               stlocV_2
 ldlocV_2               ldlocV_2
 brfalse                brfalse
 nop                    nop
 ldc.i41                ldc.i41
 stlocV_1               stlocV_1
 ldarg0                 ldarg0
 stlocV_3               stlocV_3
 ldlocV_3               ldlocV_3
 brfalse                brfalse
 nop                    nop
                     +  ldc.i4100
                     +  stlocV_4
 ldc.i42                ldc.i42
 stlocV_1               stlocV_1
 nop                    nop
 nop                    nop
 br                     br
 nop                    nop
 ldc.i42                ldc.i42
 stlocV_1               stlocV_1
 ldarg1                 ldarg1
-stlocV_4            +  stlocV_5
-ldlocV_4            +  ldlocV_5
 brfalse                brfalse
 nop                    nop
 ldc.i45                ldc.i45
 stlocV_1               stlocV_1
 nop                    nop
 br                     br
 nop                    nop
 ldc.i44                ldc.i44
 stlocV_1               stlocV_1
 nop                    nop
 nop                    nop
 ldlocV_1               ldlocV_1
-stlocV_5            +  stlocV_6
 br                     br
-ldlocV_5            +  ldlocV_6
 ret                    ret
```



```
 nop                    nop
 ldarg0                 ldarg0
 ldarg1                 ldarg1
 and                    and
 stlocV_c               stlocV_c
 ldlocV_c               ldlocV_c
 stloc                  stloc
 ldloc                  ldloc
 brfalse                brfalse
 nop                    nop
 ldc.i41                ldc.i41
 stlocV_i               stlocV_i
 ldarg0                 ldarg0
 stloc                  stloc
 ldloc                  ldloc
 brfalse                brfalse
 nop                    nop
                     +  ldc.i4100
                     +  stlocV_y
 ldc.i42                ldc.i42
 stlocV_i               stlocV_i
 nop                    nop
 nop                    nop
 br                     br
 nop                    nop
 ldc.i42                ldc.i42
 stlocV_i               stlocV_i
 ldarg1                 ldarg1
 stloc                  stloc
 ldloc                  ldloc
 brfalse                brfalse
 nop                    nop
 ldc.i45                ldc.i45
 stlocV_i               stlocV_i
 nop                    nop
 br                     br
 nop                    nop
 ldc.i44                ldc.i44
 stlocV_i               stlocV_i
 nop                    nop
 nop                    nop
 ldlocV_i               ldlocV_i
 stloc                  stloc
 br                     br
 ldloc                  ldloc
 ret                    ret
```

Since the size of an arbitrary method is arbitrarily large the previously described labels could become arbitrarily large as well. Therefore we also use a hash function to shorten the labels to a fixed size. The labels for the Test function and IL graph that we demonstrated earlier then become:

```
Test(bool, bool)                                        hash(Nested Merkle Tree Nodes)
  Test(bool, bool)/hash()                               hash(GREEN nodes)
  Test(bool, bool)/hash(ldarg0/ldarg1/and/stlocV_c/ldlocV_c/brfalse#true) hash(BLUE nodes)
  Test(bool, bool)/hash(ldarg0/ldarg1/and/stlocV_c/ldlocV_c/brfalse#false) hash(PINK nodes)
  Test(bool, bool)/hash({ green and blue prefixes of yellow })      hash(YELLOW nodes)
  Test(bool, bool)/hash({ green and pink prefixes of brown })       hash(BROWN nodes)
  Test(bool, bool)/hash({ green and pink prefixes of purple })      hash(PURPLE nodes)
```

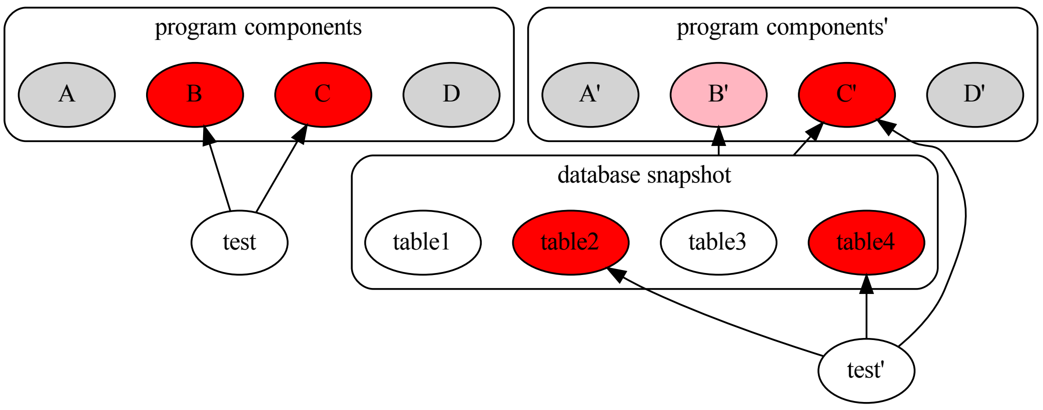
7.6. Flexible Prune Specifications

The input pruning logic in Bazel is currently done via an unused-inputs-list. This approach is limited by only expressing the set of known inputs that were unused. When additional inputs, such as code modifications, are added this, means the entire test scenario is re-selected for testing. For many use cases, such as programming languages, it is safe to argue that these new inputs only matter when existing inputs were changed to reference them. Therefore we propose and will implement that Bazel accepts a more flexible prune specification in the known format of .gitignore files. This adds support for patterns such as wildcards and negations. Using these we can reverse the unused list to a used list for certain file types by, for example, ignoring all *.dll's and negating only the covered files and code segments. Additionally, because Git works with Merkle Tree algorithms, we suspect that applicable algorithms for processing this format are readily available. Implemented correctly the semantics of existing unused-inputs-list files will remain the same.

We must also implement support to express inclusions of only portions of a file. For this we plan to extend the gitignore format to accept additional identifier segments after a # indicating subfile Merkle Tree identifiers.

7.7. Database Snapshots

Our test scenarios start by preparing the test environment with seed data. Because this seed data goes through many different pipelines of the application, this process actually causes a lot of code coverage. The actual test scenario itself however, only relies on a very small portion of the effects that this seed data has on the application. We therefore implement a database snapshot that takes a checkpoint of the application state and persists it as build system values. That way a separate build system task covers the code to reach this checkpoint, and the test task only depends on the snapshot itself. We can then further prune the dependency on this snapshot by only keeping the truly loaded database tables as a dependency to the test task. To illustrate this a schematic representation is given below.



While implementing this feature we initially gained very little benefits from it. This was due to the fact that our database snapshot contained non-deterministic data that differed every time the snapshot was created. This data mainly consisted of timestamps and random unique identifiers. We resolved this by removing some unnecessary timestamps from our snapshot, and setting other timestamps to a fixed value in case of testing. For the random identifiers we chose to implement a fixed seed, which causes the identifiers to be deterministic as long as they were being assigned orderly. Thanks to pruning we however only had to fix the cases that were actually part of the covered information.

7.8. Limitations

7.8.1. JavaScript and TypeScript

To instrument frontend code, which is written in TypeScript, we planned to use the Istanbul (Coe, 2018) instrumenter. Istanbul also keeps track of hit locations, similar to Coverlet, so we expected that the implementation would be trivial. During engineering we soon encountered some fundamental issues with the design of the JavaScript language in combination with code coverage based test selection. Large parts of the JavaScript code base were always being covered whilst also being very prone for modifications. This resulted in very poor performance which caused us to abandon the implementation of this language for our final work.

In future work we will discuss some ideas on how we could overcome these issues. First we however describe the difference between JavaScript and C# that causes these issues.

C# is a compiled language that resolves fully qualified names of classes and methods during compile time. When a method is invoked at runtime it uses this linked information to directly invoke nested methods. This way the using statements at the top of each C# file are abstracted away, and only actual method bodies that are invoked, are covered. JavaScript on the other hand, relies on its runtime to dynamically stitch the different parts of the codebase together. Import and export statements declare new classes and methods, similar to C#. For JavaScript however the function bodies do not link against fully qualified names. To know where nested function invocations are resolved from, the top level scope of a file must be interpreted first, which is done when the JavaScript application is loaded. This means that each JavaScript file will result in at least some coverage for each test. This would not be an issue if the code in this scope was very stable. However, because we declare imports and exports here it is actually very likely to change. A programmer only has to invoke one new method from an arbitrary function in a JavaScript file, and a new import is added at the file scope level, resulting in a covered change. This fundamental difference between C# and JavaScript makes it very hard to apply code coverage based test selection to JavaScript code.

7.8.2. Limitations of Code Coverage

Code Coverage based test selection is safe when program modifications that happen outside the covered scope can not influence program flow. A programming language can however implement different features that should be considered with care. A notable example here is virtual function calls and their problem is best demonstrated with an example.

```
abstract class Fruit {
    public virtual int Price() => 1;
}

class Apple : Fruit {
    // public override int Price() => 2;
}

void Test(Fruit fruit) {
    Assert(fruit.Price() == 1);
}
```

The basic test above will succeed for Apple. However by overriding the implementation of Price, as demonstrated in the comment, we can cause the test to fail without touching the previously covered program flow.

A solution to this problem that we found in literature was to connect a virtual function call to each possible override (Harrold et al., 2001). In our approach this would mean that instrumentation of a virtual function call requires a holistic view of all implemented overrides. Because our approach instruments each project individually this was not a feasible constraint. Another downside of this approach is that the static set of *all possible overrides* is often larger than the actual covered overrides during execution. In our example this would mean that introducing a new kind of fruit required the Apple test to be evaluated again as well.

A possible solution that we came up with was to simply implement all possible overrides during instrumentation. The empty Apple class would then be transformed into something like:

```
class Apple : Fruit {
    public override int Price() => base.Price();
}
```

Because this generated override will result in code coverage this fits perfectly in our coverage based architecture. In case a programmer would implement their own override this would cause the covered code change, hence resulting in test selection. It does however drastically complicate the instrumentation step because we must analyze which virtual functions exist without override for each individual class.

7.8.3. *Generalization of Information Coverage*

```
void Test {
    Assert(!File.Exists("somefile.txt"));
}
```

We can generalize the previously described virtual function problem as recognizing that a piece of absent information may become part of the covered information when it becomes present. In a simple form a similar problem could occur when a test tries to read a file but graciously continuous when its not present. An example of such test is shown above. When *somefile.txt* becomes present it was not part of the previously covered information. It does however cause the test to fail. To support these scenarios we must introduce a way to express which information will influence our result when it becomes available. Besides tracing the covered information during a task execution we therefore also introduce trace data for information keys that were attempted to be covered but turned out to not exist (yet).

When regarding the fact that a piece of information is not present as a piece of information itself we could argue that we still only have to trace the coverage of information. When tracing value structure and dependency coverage as hashsets this only works when the set of information that could exist, but currently does not exist, is known. This is required because we must be able to answer if a piece information was not present in a given value structure trace before. When everything is represented as a hash however, there is no way to find if a piece of information is missing, or has a different value. Therefore we must encode the fact that it is missing as a hash entry itself when tracing the value structure. This is similar to our proposed solution in the previous section of introducing additional coverable information by creating an empty override. The map and merkle tree variants of traces do not face the same challenge. Here we can simply query the data structure if a given key exists to find out if it is currently a non existing piece of information. An information coverage trace can then also contain keys where the value hash is empty, which allows build system task to express dependencies on currently missing information.

Sometimes the information that we try to resolve is not definable by a single key. We could for example try to read all **.json* files from a directory or ZIP archive. In this case simply adding a new JSON file would impact our execution and thus, being modification traversing, possibly our results.

To facilitate complex dependencies on currently missing pieces of information or collections of information matching a certain pattern we propose the following model enhancement.

Information dependencies like this can be expressed in the form of glob patterns similar to .gitignore files [1]. We could also consider a directory that is globbed to be part of the covered information. Adding files to the directory causes the directory itself to change leading to the invalidation of a dependency.

[1] To read more about .gitignore files visit https://git-scm.com/docs/gitignore

Chapter 8

Results

A proof of concept has been implemented to demonstrate the feasibility of the previously described architecture. This prototype currently implements the following features:

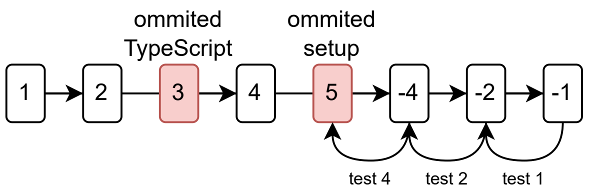
- Instrumentation of C# code at the granularity of methods and branches.
- Providing value structure traces for the instrumented C# binaries.
- Measuring C# code coverage during test execution resulting in dependency coverage traces.
- Creating a database snapshot to checkpoint the seed phase of a test. This snapshot is also represented by a value structure trace.
- Modifying runtime behavior to produce a dependency coverage trace for the loaded parts of a database snapshot.
- Pruning entire dependencies of a test task.
- Pruning subsets of test task dependencies using the value structure and dependency coverage traces for both C# and the database snapshot.

It therefore, notably, does not implement a solution to the virtual function problem and other not coverable modifications as described in section 7.8.2.. We also decided not to include JavaScript and TypeScript support due to the previously described fundamental issues. Because of this we do not include TypeScript modifications in our results since they would skew the results for the modification types that we did implement.

8.1. Experimental Setup

To validate our work we have conducted experiments on the private codebase of AFAS Software. This codebase consists of a large monorepo that implements a low code platform consisting of over 500 C# projects. We required various modifications to the codebase for our experiments which we implemented on top of the latest version of the codebase. To validate our test selection method we then recreated a git history of the codebase between 11-04-2023 and 03-05-2023 by reverting individual commits, excluding our previous modifications that were done to facilitate our experiment. After each revert we validated the codebase would still build, occasionally resolving conflicts that were caused by our modifications. This process ultimately resulted in 108 individual commits that we used as real world codebase alterations for our experiment.

Git makes it easy to traverse its commits in reverse because each commits refers to its previous commit. Because of that we chose to revert the commits in reverse historical order starting with the newest commit. This resulted in a reverse chain of commits that ended in the oldest commit. This was actually convention for us, because we could now traverse the commits in forward historical order, starting with the oldest. We measured, for each change between commits, which integration test scenarios were selected for testing while applying information coverage based test selection.



8.2. Hardware

All experiments were conducted on a Dell Precision 5550 laptop with the following specs:

- CPU: Intel Core i7 10875H
- GPU: NVIDIA Quadro T2000
- RAM: 32GB
- Storage: Micron 2300 NVMe 1024GB

8.3. Experiments

To understand the impact of the different techniques that we applied we ran the full experiment five times:

- Without any information coverage based selection
- Without database snapshotting, applying method coverage based test selection
- Without database snapshotting, applying branch coverage based test selection
- With database snapshotting and method coverage based test selection
- With database snapshotting and branch coverage based test selection

The prior test selection method, that only used existing dependency graph information, would either select all integration test scenarios or none. If any potential dependency of a test was modified, all tests were selected. If the codebase modification was outside of this scope, none were selected. This could happened occasionally when the modification only affected other tests, or modified non code files such as documentation.

In early trials of the experiment we were already seeing significant gains from only applying C# method coverage based test selection. In a given run of 29 codebase revisions we managed to reduce a 93% selection rate down to 42%. The table below shows these results. The first line contains baseline selection rates which are either 0 or all 46 tests. The second line contains the selection rate with the method coverage based selection feature active.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
baseline	46	46	46	46	46	46	46	46	46	0	46	46	46	46	46	46	46	46	46	46	46	46	46	46	46	0	46	46	46
selection	0	46	0	0	0	0	0	46	46	0	46	19	0	0	0	46	46	1	0	46	1	25	46	46	46	0	1	46	0

Our initial results already gave various interesting insights. Where almost all changes used to result in a full test run before, code coverage based test selection caused 0 tests to be selected in 41% of the changes. The second largest set of changes were still those that affected every single test case. This was either due to them being a refactor of some commonly used low level concepts or because they covered processes in the initial database seed that every test used.

To optimize for tests that were being selected due to coverage of the seed process, we implemented a way to differentiate between the seed process and the actual test runtime. Our approach consisted of subdividing the seed process and test execution into separate build system tasks. Here the seed process produced a database snapshot value, for which the test execution task could trace coverage independently from code coverage. This architecture drastically reduced the code coverage footprint of test execution, but introduced additional dependencies in the form of database snapshot coverage. Our initial results using the snapshotting technique were very disappointing due to volatlity in the database snapshot itself. In section 7.7. we already discussed these problems, and our solution in more detail.

Finally, to reduce test case selection even further, we also optimized the precision of the code coverage measurements by implementing branch coverage based value structure and dependency coverage traces. This higher resolution coverage data allowed us to safely exclude test cases that did cover a modified method, but only covered unmodified branches of that method. In the next section the benefits of this improved resolution are discussed in more detail.

8.4. Results

	Selection Rate	Over Baseline	Over Previous	Over No Snapshot
Baseline	78,7%			
Method Coverage	29,6%	37,7%		
Branch Coverage	28,9%	36,7%	97,3%	
Method + Snapshot Coverage	22,0%	28,0%	76,2%	74,2%
Branch + Snapshot Coverage	21,3%	27,1%	96,7%	73,7%

measured test selection rates

The findings of our experiments are shared in the table above. The baseline selection rate for our final experiment was 78.7%, which is higher than that of the initial testing rounds. Looking at the selection rate improvement of only applying method coverage based test selection, we see an improvement of 62.3% relative to the baseline. More fine grained branch coverage only adds one more percent to this, which, relative to method coverage, optimizes selection rate by 2.7%. When enabling the database snapshotting feature to reduce method coverage we see a 25.8% improvement. For branch coverage this improvement is 26.3%. With all features enabled our experiments show us a baseline improvement of 72.9% where just over 20% of test cases are still selected.

8.5. Threats to Validity

8.5.1. Internal Validity

Due to the limitations that we discussed in detail in section 7.8.2. our implementation is not complete with regards to catching all fault revealing test cases. Additionally, because our test set only contains successfully tested commits to the mainline codebase, our experiments tell us nothing about the amount of fault revealing test cases that we are potentially missing. Since the limitations we discussed only occur under very strict circumstances, we do not expect them to occur very often, if at all. Further research is however required to verify this in a production setting.

Because the runtime of a single test in our test set takes longer than a minute the additional overhead of instrumentation, coverage collection and processing during test selection are neglectable. For codebases where test task duration is lower, the overhead of our method will be higher. As a result of our implementation in the Bazel build system, and the way that this is designed, it became very difficult to measure the true performance implications during our tests. We therefore do not claim net time savings, but discuss our results objectively as test selection rates.

Another threat to internal validity is *maturation*. During implementation and experimentation we found various issues, such as non determinism in snapshots and code paths that were always being covered. By fixing these issues we influenced the results of our experiments. We do however think that these steps were required to demonstrate the true potential of coverage based test selection, and that others who apply the technique will be looking for these optimizations as well. Coverage based test selection even functions as a means of detecting these opportunities.

When considering the *history* and *sampling bias* threats to validity this could very well apply to our setup. Because we only selected a consecutive sequence of commits in time we may have selected a set that does not represent average activity well. The work being done in this limited time span could, for example, be skewed towards a specific program part for which test case selection diverges from the mean. After manually reviewing the software revisions, we are confident that they represent a diverse set of changes. Additionally, the results of our initial experiments, and those of our final results, do not diverge much. More research will however be needed to verify the results on a larger dataset.

8.5.2. External Validity

When considering the extent in which we can generalize the results of our study, we should note that the results are heavily influenced by: codebase size, changeset size and coverage per test task. Additionally, the maturity and structure of a codebase are important as well, since they influence the likelihood that changesets contain changes to commonly covered code. These so called hot paths tend to change less frequently when a codebase matures. Additionally, developers could identify code that often invalidates test results, and restructure the codebase accordingly. Because of these unpredictable dependencies we believe that it is very unlikely that another instantiation of our experiment will yield the same results. It is, however, of greater importance that our proposed methods for implementing information coverage in a build systems proved functional. Further research will be required to get a better understanding of the performance improvements of information coverage based test selection in a generalized setting.

Chapter 9

Conclusion

This thesis has explored the applicability of code coverage based test selection in a modern industry environment. By researching existing work we found that code coverage based test selection has been extensively researched in the past. Unfortunately these methods have not been widely adopted by the industry. As a contribution towards this adoption we extend the formal model of build systems with additional concepts to facilitate code coverage based test selection. The first of these generic concepts is a value structure trace that allows the build system to track changes on a more fine-grained level than files, such as coverable code segments. The second concept we introduce are dependency coverage traces that allow the build system to express the utilized subset of a value (identified by its value structure trace) when a build task is executed. Finding an empty intersection between a dependency coverage trace and the modified subset of a value structure trace allows us to determine that a build system task does not depend on anything that was modified. Because test execution is resembled by a build system task, this reduces test case selection rates.

Rothermel et al. defined this process as finding change traversing test cases and proved that this heuristic is safe. We however discovered various variations of code modifications that could cause a test to fail without being part of the baseline code coverage. A notable example of this is given in section 7.8.2. and involves overriding a virtual function. Because of this we contribute the concept of information coverage as a generalization of code coverage. Information coverage enables us to express the coverage of any piece of information that might change and is used during test execution. This allows the introduced trace types to express changes to any piece of information and track dependencies accordingly. We propose a solution to the virtual override problem by allowing dependencies on absent pieces of information.

In order to verify our model enhancements in an industry setting we have implemented a prototype on top of the Bazel build system. This approach allowed us to leverage much of the existing architecture for change tracking and incremental building. The additional features we implemented were a way to express value structure traces in the format of Merkle Trees and dependency coverage traces in the form of hit locations. This proof of concept does not implement solutions to the previously mentioned limitations of code coverage. It does however implement the data coverage solution using the same trace mechanics for a database snapshot.

With the working implementation of information coverage based test selection we conducted various experiments to measure the impact of our methods. We saw code coverage based test selection reduce the rate of integration tests to almost one third of the baseline method. After combining this approach with database snapshot coverage heuristics we managed to reduce this selection rate even further by one quarter. Interestingly, we found that improving the granularity of code coverage from methods to branches had very little impact.

9.1. Research Questions

These results allow us to answer our research questions as follows:

9.1.1. How effective is Code-Coverage based test selection on integration-test scenarios?

We found that code coverage based test selection can be effective on integration-test scenarios. Our results show that the algorithm, when measured on the granularity of methods, yields an improvement of 62.3% over the incremental build graph algorithm. Interestingly, increasing the granularity of coverage measurements to branches only improves this by one percent to 63.3%.

9.1.2. Can we generalize to Information-Coverage and does it improve our efficiency?

We found that substituting program runtime for a database snapshot, and applying data coverage heuristics on it, improved test case selection rates by an additional 25.8% when combined with method covarage and 26.3% when combined with branch coverage. Depending on the deterministic nature of an application, a database snapshot can actually work counter productive when it contains non deterministic data each time it is created. Resolving this required significant investments in changing the application logic to be deterministic for any data that was covered during test scenarios. This additional overhead imposed on developers, and the relatively low benefits of data coverage make it less worthwhile to implement and maintain.

We also discovered that for various codebase modifications, traditional code coverage is not enough. Therefore information coverage based test selection is actually a necessity for an implementation to be complete. We conclude that we were able to implement data coverage as another instance of information coverage. Data coverage also improved the efficiency of test selection, albeit with significant investments in program architecture. Additionally we also conclude that our generalization to information coverage is actually required to fully express change traversing tests using coverage data.

9.1.3. Do our methods yield good enough results for the industry towards changing its testing strategy?

An important insight from our study is that code coverage alone is not enough for a complete implementation of test selection. More work is required to implement and research the limitations that we discovered, which is crucial for a safe industry wide adoption. Our findings do show that information coverage based integration test selection can be successfully deployed in an industry setting. Purely applying code coverage heuristics already reaches significant improvements over dependency graph based test selection.

Due to the nature of integration tests however, we found that they are very likely to cover a big chunk of the codebase. We saw many program components that were being covered during the startup of the program, which caused a large coverage footprint for every test. This resulted in inefficient to testing of modifications to these program parts since they caused all tests to be selected. Improving this rate by reducing coverage required significant investments in test infrastructure. In some cases we managed to reduce the coverage by purely changing the implementation. Another solution we implemented for this are database snapshots that function as a checkpoint of application state and allow us to substitute large parts of code coverage for coverable data.

With these findings we conclude that the industry can shift towards integration testing over unit testing using our build system enhancements. This does however require that the code coverage per integration test is small, which may require significant investments in test setup to be reduced. Additionally we demonstrate that, with minimal investments, code coverage based test selection reaches significant results for existing integration tests without any modification.

Chapter 10

Future Work

We will end this work with some possibilities for future work.

10.1. Validating the algorithm in production

The most important next step is to validate the build system enhancements in a real continuous integration (CI) setup. Our results currently only include already successful codebase revisions. Therefore they do not measure anything about the completeness of the method. By running the algorithm in a real world CI setup we can get a better understanding of how likely the current implementation is to miss a faulty test. This research will also be able to produce a list of fault revealing codebase modifications that do not surface with traditional code coverage based test selection. Another important next step is to validate our enhancements against other codebases. This will give a better understanding of how well our methods generalize. To facilitate research in real world CI setups of various codebases we open source our current implementation in Bazel.

10.2. Lower granularity

Our results show only one percent increase in performance when implementing branch coverage compared to method coverage. This raises the question what impact it would have if we further reduced the granularity of coverage to class or namespace levels of sourcecode. Reducing granularity could bring performance improvements as well as consume less memory. Additionally, when considering the coverage at the granularity of classes, we see various benefits with regards to the limitations we discussed. A virtual function override for example, is always part of a class. Introducing the override will then mutate the class that may or may not be already part of the measured class coverage. This depends on whether an object of that class is created during test execution, which we can safely measure.

10.3. Information Coverage for Build Systems

When applied properly, we can use our information coverage algorithm as a way to minimize the entire build system, not limited to test tasks. Build system tasks of any kind receive information as inputs and potentially use only a subset of the information. Compilation of .NET projects, for example, already uses the concept of reference assemblies. These assemblies only contain the public signatures of functions without any implementation, similar to .h files for C(++). This is because this is the only information that the compiler needs to compile another project that references these signatures. These reference assemblies are less volatile than the actual assemblies containing implementations. This in turn improves the minimality of the build system. With our information coverage approach we could replace these reference assemblies with a subset of the information the actual assemblies provide. With some caveats this allows us to even further optimize for minimality by only maintaining the function signatures that were actually used during compilation as action inputs, as opposed to all the function signatures.

10.4. Dynamic programming languages

While implementing the same heuristics for code modifications to our TypeScript codebase we encountered that the dynamic symbol resolution of this language caused the set of covered code code to often intersect with changed code. Ultimately this rendered the implementation for JavaScript and TypeScript out of reach for now. We can however imagine ways to bring change detection of TypeScript functions closer to that of our C# implementation. This would require some form of symbol resolution to occur when determining content hashes of function bodies. An interesting approach to symbol resolution is discussed by GitHub employees (Clem & Thomson, 2021) who utilize so called Stack Graphs (Creager & van Antwerpen, 2022) to incrementally track symbol resolution information. A benefit of their toolkit is the use of Tree Sitter, which is a universal parser that can support any language. Some of their work could potentially be used to properly determine content hashes for dynamic languages.

References

Ali, N. B., Engström, E., Taromirad, M., Mousavi, M. R., Minhas, N. M., Helgesson, D., … Varshosaz, M. (2019). On the search for industry-relevant regression testing research. *Empirical Software Engineering*, *24*(4), 2020–2055.

Baker, C. L. (1957). *Digital computer programming; a book review* (Tech. Rep.). RAND CORP SANTA MONICA CALIF.

Bazel. (2015). (`https://bazel.build/`)

Bell, J. T. (2001). Extreme programming. *Thinking for Innovation*.

Beszédes, A., Gergely, T., Schrettner, L., Jász, J., Langó, L., & Gyimóthy, T. (2012). Code coverage-based regression test selection and prioritization in webkit. In *2012 28th ieee international conference on software maintenance (icsm)* (p. 46-55). doi: 10.1109/ICSM.2012.6405252

Brito, G., Terra, R., & Valente, M. T. (2018). Monorepos: a multivocal literature review. *arXiv preprint arXiv:1810.09477*.

Brousse, N. (2019). The issue of monorepo and polyrepo in large enterprises. In *Companion proceedings of the 3rd international conference on the art, science, and engineering of programming* (pp. 1–4).

Clem, T., & Thomson, P. (2021). Static analysis at github: An experience report. *Queue*, *19*(4), 42–67.

Coe, B. (2018). *Istanbul.* (`https://istanbul.js.org/`)

Creager, D. A., & van Antwerpen, H. (2022). Stack graphs: Name resolution at scale. *arXiv preprint arXiv:2211.01224*.

de Oliveira Neto, F. G., Ahmad, A., Leifler, O., Sandahl, K., & Enoiu, E. (2018). Improving continuous integration with similarity-based test case selection. In *Proceedings of the 13th international workshop on automation of software test* (pp. 39–45).

Dijkstra, E. W., et al. (1970). *Notes on structured programming.* Technological University, Department of Mathematics.

Ebert, C., Gallardo, G., Hernantes, J., & Serrano, N. (2016). Devops. *Ieee Software*, *33*(3), 94–100.

Elbaum, S., Gable, D., & Rothermel, G. (2001). The impact of software evolution on code coverage information. In *Proceedings ieee international conference on software maintenance. icsm 2001* (pp. 170–179).

Elbaum, S., Rothermel, G., & Penix, J. (2014). Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the 22nd acm sigsoft international symposium on foundations of software engineering* (pp. 235–245).

Elmendorf, W. (1967). Evaluation of the functional testing of control programs. *IBM, Pughkeepsie, New York*.

Elmendorf, W. R. (1969). Controlling the functional testing of an operating system. *IEEE Transactions on Systems Science and Cybernetics*, *5*(4), 284–290.

Elsner, D., Hauer, F., Pretschner, A., & Reimer, S. (2021). Empirically evaluating readily available information for regression test optimization in continuous integration. In *Proceedings of the 30th acm sigsoft international symposium on software testing and analysis* (pp. 491–504).

Engström, E., & Runeson, P. (2010). A qualitative survey of regression testing practices. In *International conference on product focused software process improvement* (pp. 3–16).

Feldman, S. I. (1979). Make—a program for maintaining computer programs. *Software: Practice and experience*, *9*(4), 255–265.

Fischer, K. F. (1977). A test case selection method for the validation of software maintenance modifications.

Fowler, M., Highsmith, J., et al. (2001). The agile manifesto. *Software development*, *9*(8), 28–35.

Hans Dockter, E. W. M. P., Wayne Caccamo. (2019). *The developer productivity engineering handbook*. Gradle. (`https://gradle.com/wp-content/uploads/2019/09/Developer-Productivity-Engineering-eBook.pdf`)

Harrold, M. J., Jones, J. A., Li, T., Liang, D., Orso, A., Pennings, M., … Gujarathi, A. (2001). Regression test selection for java software. *ACM Sigplan Notices*, *36*(11), 312–326.

Henderson, F. (2017). Software engineering at google. *arXiv preprint arXiv:1702.01715*.

Ivanković, M., Petrović, G., Just, R., & Fraser, G. (2019). Code coverage at google. In *Proceedings of the 2019 27th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering* (pp. 955–963).

Joe Morris, M. J. (2017). *Live unit testing in visual studio 2017 enterprise.* (`https://devblogs.microsoft.com/visualstudio/live-unit-testing-in-visual-studio-2017-enterprise/`)

Maudoux, G., & Mens, K. (2018). Correct, efficient, and tailored: The future of build systems. *IEEE Software*, *35*(2), 32–37.

McCracken, D. D. (1957). *Digital computer programming*. John Wiley & Sons, Inc.

Merkle, R. C. (1979). *Secrecy, authentication, and public key systems.* Stanford university.

Meszaros, G., Smith, S. M., & Andrea, J. (2003). The test automation manifesto. In *Conference on extreme programming and agile methods* (pp. 73–81).

Mokhov, A., Mitchell, N., & Peyton Jones, S. (2018). Build systems à la carte. *Proceedings of the ACM on Programming Languages*, *2*(ICFP), 1–29.

Mukherjee, R., & Patnaik, K. S. (2021). A survey on different approaches for software test case prioritization. *Journal of King Saud University-Computer and Information Sciences*, *33*(9), 1041–1054.

Ojdanić, M. (2022). Change-aware mutation testing for evolving systems. In *Proceedings of the 30th acm joint european software engineering conference and symposium on the foundations of software engineering* (pp. 1785–1789).

Overeem, M., et al. (2022). *Evolution of low-code platforms* (Unpublished doctoral dissertation). Utrecht University.

Petrović, G., & Ivanković, M. (2018). State of mutation testing at google. In *Proceedings of the 40th international conference on software engineering: Software engineering in practice* (pp. 163–171).

Petrovic, G., Ivankovic, M., Kurtz, B., Ammann, P., & Just, R. (2018). An industrial application of mutation testing: Lessons, challenges, and research directions. In *2018 ieee international conference on software testing, verification and validation workshops (icstw)* (pp. 47–53).

Piwowarski, P., Ohba, M., & Caruso, J. (1993). Coverage measurement experience during function test. In *Proceedings of 1993 15th international conference on software engineering* (pp. 287–301).

Potvin, R., & Levenberg, J. (2016). Why google stores billions of lines of code in a single repository. *Communications of the ACM*, *59*(7), 78–87.

Rothermel, G. (1996). *Efficient, effective regression testing using safe test selection techniques*. Clemson University.

Rothermel, G., & Harrold, M. J. (1997). A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *6*(2), 173–210.

Schalkwijk, J. (2011). Smart regression testing with the t2 testing framework.

Solarin-Sodara, T. (n.d.). *Coverlet.* (`https://github.com/coverlet-coverage/coverlet`)

Yoo, S., & Harman, M. (2012). Regression testing minimization, selection and prioritization: a survey. *Software testing, verification and reliability*, *22*(2), 67–120.

Young, G. (2012). *Mighty moose.* (`https://gregfyoung.wordpress.com/tag/mighty-moose/`)