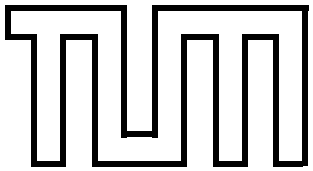


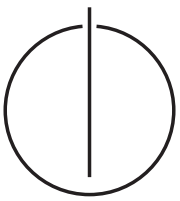
FAKULTÄT FÜR INFORMATIK

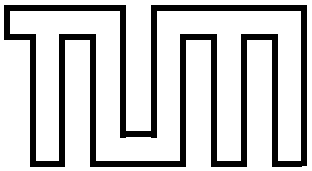
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Masterarbeit in Informatik

Empirische Untersuchung der Effektivität von Testpriorisierungsverfahren in der Praxis

Jakob Rott





FAKULTÄT FÜR INFORMATIK

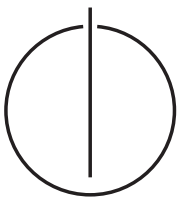
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Masterarbeit in Informatik

Empirische Untersuchung der Effektivität von Testpriorisierungsverfahren in der Praxis

Empirical Study on the Effectiveness of Test Prioritization Techniques in Practice

Bearbeiter:	Jakob Rott
Aufgabensteller:	Prof. Dr. Dr. h.c. Manfred Broy
Betreuer:	Rainer Niedermayr
	Dr. Elmar Jürgens
Abgabedatum:	15. Februar 2019



Ich versichere, dass ich diese Masterarbeit in Informatik selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 15. Februar 2019 Jakob Rott

Zusammenfassung

Automatisierte Tests sind ein zentrales und unverzichtbares Qualitätssicherungsinstrument bei der kontinuierlichen Weiterentwicklung von langlebiger Software. Die Tests werden im Rahmen der Continuous Integration idealerweise bei jedem Push (oder Commit) ausgeführt und helfen somit, Fehler frühzeitig aufzudecken. Bei wachsender Systemfunktionalität und damit einer zunehmenden Anzahl an automatisierten Testfällen steigt jedoch die Ausführdauer der gesamten Test-Suite. Bei großen Systemen erreicht die Ausführdauer oft die Größenordnung von Tagen oder gar Wochen. Somit ist es für diese Systeme selbst unter dem Einsatz von Parallelisierung nicht mehr möglich, alle Testfälle bei jedem Push auszuführen. Als Folge ist Continuous Integration nicht mehr möglich und die Zeit zwischen dem unbeabsichtigten Einbau eines Fehlers durch eine Code-Änderung und dessen Erkennung steigt. Eine mögliche Lösung ist es, die regelmäßige Ausführung der Tests auf eine Teilmenge zu beschränken, die pro Testlauf in Abhängigkeit von den durchgeführten Code-Änderungen neu gewählt wird. Um die Teilmenge für jeden Testlauf neu zu bilden, kommen Testselektionstechniken zum Einsatz. Auszuführende Test können zudem durch eine Priorisierung in ihrer Ausführungsreihenfolge so umsortiert werden, dass ein gewünschtes Ziel möglichst schnell erreicht wird. Dies kann zum Beispiel die zeiteffiziente Abdeckung verschiedener Systemteile sein.

Bisherige Studien wurden mit unterschiedlichen Evaluierungstechniken (echte Fehlschläge historischer Testausführungen vs. durch Mutationen verursachte Fehlschläge) und für verschiedene Studienobjekte durchgeführt, sodass eine Generalisierbarkeit dieser Studien nicht gewährleistet ist. Hierbei wurde die Zeit bis zum ersten Testfehlschlag (Time to Failure) als ausschlaggebende Metrik betrachtet.

In dieser Arbeit wurden Testpriorisierungsalgorithmen sowohl auf in der Vergangenheit fehlgeschlagene Builds angewendet, als auch in einem Benchmark, der mit Daten aus Mutationsanalysen arbeitet. Eine Teilmenge der Studienobjekte konnte durch beide Methoden untersucht werden. In beiden Analysen zeigten sich durch den Einsatz der an der Forschungsgruppe entwickelten Strategie die niedrigsten Time to Failure-Werte. Bei zwei von drei Projekten, von denen die Ergebnisse aus beiden Analysen verglichen wurden, zeigten sich keine auffälligen Unterschiede in der Strategieperformanz. Anhand eines Projektes wurde untersucht, ob sich ältere Testausführungsinformationen für die Nutzung im Selektions- und Priorisierungsverfahren eignen. Dabei wurde der fehlgeschlagene Test in fünf von 15 Fällen nicht mehr ausgeführt, die Time to Failure blieb allerdings niedrig. Weiterhin wurde die Berechnung der Testmenge und -reihenfolge um Optionen erweitert, die die vorgeschlagene Testausführung an Beschränkungen eines testausführenden Systems ausrichtet. Die Auswirkungen unterschiedlicher Konfigurationen wurden an einem Algorithmus gemessen, wobei mit erhöhter Restriktion die Time to Failure Werte leicht anstiegen.

Inhaltsverzeichnis

1 Einleitung

The purist will demand that all previously used test cases be rerun.

FISCHER ET AL., 1981 [FRC81]

In der modernen Softwareentwicklung sind kurze Releasezeiten auch bei großen Systemen üblich. Um dies zu ermöglichen, wird in vielen Entwicklungsumgebungen kontinuierlich integriert (siehe Grundlagen: CI). Das bedeutet, dass Änderungen, die Entwickler an der Software machen, häufig in die Codebasis eingepflegt werden. Um sicherzustellen, dass Funktionalitäten, die bereits funktioniert haben, nicht durch eine Änderung im Code beeinträchtigt werden, werden Regressionstests eingesetzt. Mit der Größe eines Systems wächst in der Regel auch die Ausführungsdauer der gesamten Testmenge. Eine große Anzahl qualitätssichernder Tests ist gewünscht, allerdings können sich lange Auführungszeiten im Buildprozess negativ auf den Entwicklungsprozess auswirken. In erster Linie ist es dabei nicht die Last auf den Buildservern, die stört, sondern die langen Wartezeiten der Entwickler, wenn sie wissen möchten, ob die eingecheckten Codeänderungen keinen Testfehlschlag zur Folge haben.

Werden langlaufende Test-Suiten aus dem CI-Prozess herausgenommen und nur in größeren Zeitabständen ausgeführt (zum Beispiel nächtlich, wöchentlich oder vor einem Release), so steigen sowohl der Aufwand, auftretende Fehler zu beheben, als auch die Wahrscheinlichkeit von Feldfehlern.

Um gegen dieses Problem vorzugehen, gibt es in der Literatur verschiedene Vorschläge, nicht alle Testfälle in einer beliebigen Reihenfolge auszuführen, sondern die Auführungsreihenfolge abhängig von den gemachten Codeänderungen zu gestalten und möglichgerweise auch die Testmenge zu reduzieren.

In der Praxis haben Testpriorisierungsverfahren noch nicht die notwendige Verbreitung gefunden.

1.1 Entwicklung in der Ausführungshäufigkeit von Softwaretests

Kurze Buildzeiten und damit schnelles Feedback sind gerade in Projekten, die den Grundsätzen der kontinuierlichen Integration folgen, wichtig [SB14]. Als Grund wird unter anderem auch genannt, dass die Erinnerung eines Entwicklers an eine gerade getätigte Änderungen am Programmquelltext noch frisch ist [DMB12]. Auch Rasmusson beschreibt [Ras04], dass dem Entwickler bei langlaufenden Builds zwei Möglichkeiten bleiben. Entweder das Durchlaufen des Builds wird abgewartet oder alternativ mit einer (zwischenzeitlichen) Aufgabe begonnen. Zwischenzeitlich deshalb, weil die Integration im Falle eines fehlschlagenden Builds später wieder aufgenommen und komplettiert werden muss. Dies hat einen Nachteil auf die Performance eines einzelnen Teammitglieds und kann sich auf die gesamte Entwicklergruppe auswirken [Ras04].

In der Workshoppräsentation von Niedermayr zum Paper [Rot+17] zeigte dieser auf, wie sich die Testausführung verändert hat. In modernen Projekten wird im Vergleich zu

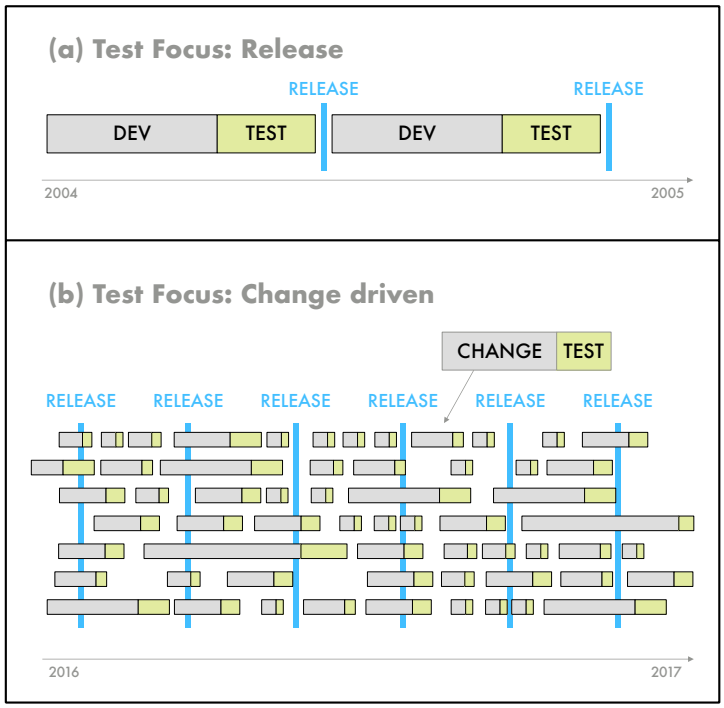


Abbildung 1.1: Der Testfokus liegt in modernen Projekten häufig nicht auf dem Release, sondern Änderungen werden kontinuierlich getestet. (Quelle: Modifizierte Präsentationsfolien [Rot+17], mit freundlicher Genehmigung von Niedermayr)

früher öfter und mehr parallel entwickelt, und Tests werden nicht nur vor dem Release, sondern auch für kleinere Änderungen am System ausgeführt (siehe Abb. 1.1).

Obwohl in GitHub-Travis-Projekten die Testdauer oft kurz ist (Median: 1 Minute) und auch die maximale Testdauer (bis 30 Minuten) nicht kritisch hoch erscheint [BGZ17a], berichten Ståhl und Bosch von Buildzeiten von über einer Stunde [SB14]. Rasmusson schreibt von Builds, die einige Stunden andauern können [Ras04].

1.2 Problemstellung

Zur Evaluation von Testselektions- und Priorisierungsverfahren finden sich in der Literatur Auswertungen, ob durch eine ausgewählte Teilmenge von Tests, weniger Mutanten entdeckt werden oder ob durch die Priorisierung von Testfällen mehr Mutanten pro Zeit im Code entdeckt werden. Es lässt sich allerdings zum derzeitigen Stand keine Veröffentlichung finden, die anhand real aufgetretener Fehler theoretisch berechnet, ob der Testfehlschlag unter Verwendung eines Priorisierungsalgorithmus schneller aufgetreten wäre. Dies ist ein praktisch relevanter Aspekt dieses Forschungsbereiches, denn ist es die Zeit bis zum Fehlschlag eines Builds, die von Entwicklern am nächsten erfahren wird und die einen agilen Arbeitsfluss behindern kann

Diese Lücke soll mit der hier vorliegenden Arbeit geschlossen werden.

1.3 Beitrag der Arbeit zur Untersuchung der Effektivität von Testpriorisierungsverfahren in der Praxis

Innerhalb der Forschungsgruppe wurde bereits an Testpriorisierungs- und Selektionsverfahren und deren Evaluation geforscht. Bevor in Kapitel 3 ausführlicher auf verwandte

Arbeiten eingegangen wird, fassen die folgenden beiden Absätze die derzeitige Situation in der Forschungsgruppe zum Benchmarking von Priorisierungsalgorithmen zusammen und heben den Beitrag der hier vorliegenden Arbeit hervor.

1.3.1 Aktueller Stand

In einer früheren Arbeit in der Forschungsgruppe [Rot18] wurde erstmalig ein Benchmark für Testpriorisierungsalgorithmen entworfen, der mit tatsächlich aufgetretenen Buildfehlschlägen als Datenquelle arbeitet. Dieser wurde unter anderem auch auf das Selektions- und Priorisierungsverfahren angewendet, das von Dreier entworfen und implementiert wurde [Dre17]. Dreier evaluierte den vorgeschlagenen Algorithmus anhand von Mutationsanlysen in verschiedenen Projekten. Es gibt keine Studienobjekte, die sowohl von Dreier als auch von Rott untersucht wurden. Die Frage, ob der entworfene Algorithmus in gleichen Projekten sowohl in Mutationsanalysen als auch bei real auftretenden Testfehlern hinsichtlich der frühen Priorisierung des Testfalls performant arbeitet, ist deshalb noch ungeklärt.

1.3.2 Beitrag der Arbeit

Die vorliegende Arbeit wendet verschiedene Priorisierungsalgorithmen in teils unterschiedlicher Konfiguration auf tatsächlich aufgetretene Testfehler an. Dafür wurden Builds mit fehlschlagenen Tests von Github Projekten identifiziert und die Integration der Priorisierungsalgorithmen nachgestellt. Zusätzlich wurde dieses Verfahren auch auf neun Builds der kommerziellen Software Teamscale angewandt.

So wurde untersucht, wie gut die Priorisierungsalgorithmen – unter der Prämisse, schnelles Feedback an Entwickler geben zu können – in realen Situationen funktionieren. Es wurde außerdem auf Störfaktoren wie veränderte Testdaten oder infrastrukturelle Fehlschäge eingegangen, die im Alltag regelmäßig vorkommen, im Studienumfeld allerdings oft außer Acht gelassen werden.

Der Ansatz aus dem Prototypen [Rot18] wurde verbessert und an die aktualisierte Version des darunterliegenden Analysetools Teamscale angepasst (siehe dazu Abschnitt 4.3). Die Daten daraus wurden untersucht und ein erneuter Benchmark durchgeführt.

Zur Bewältigung der Fragestellungen wurden einige Java-Werkzeuge implementiert, wie zum Beispiel das Programm `TestPrioritizationBenchmark`, das Testausführungslisten abruft und relevante Daten für den Benchmark berechnet, und der `MutationResultDataPersister`, der Ausgabedaten des Mutation-Based-Benchmarksystems von Niedermayr parst und in einer Datenbank ablegt.

Anhand von Beispielprojekten wurden die Ergebnisse aus einer mutationsbasierten Analyse mit den Ergebnissen aus dem Real-World-Benchmark verglichen.

2 Grundlagen / Begriffsdefinitionen

Die folgenden Absätze beschreiben Begrifflichkeiten, die zum Verständnis der Arbeit beitragen. Sie stammen überwiegend aus dem Gebiet der Softwaretechnik und des Testens.

Continuous Integration Continuous Integration (CI) ist ein Begriff in der Softwareentwicklung, der von Grady Booch eingeführt wurde [Boo91]. Folgt man den Grundsätzen der CI, so wird von einem Entwickler nicht über einen längeren Zeitraum hinweg isoliert implementiert und dann eine große Änderung an das Versionkontrollsystem (VCS) übergeben, sondern es werden viele kleinschrittige Änderungen regelmäßig in das VCS integriert. Dabei sollen unter anderem auch Regressionstests ausgeführt werden, die überprüfen, ob durch die veränderten Codestellen (Code-Delta) andere Funktionalitäten des Programms, die in einer früheren Version fehlerfrei waren, ungewollt funktionsuntüchtig wurden. In einem weit verbreiteten Artikel schreibt Martin Fowler, dass jede an der Entwicklung beteiligte Person ihre Arbeit mindestens einmal am Tag integrieren sollte [FF06]. Eine Überblicksstudie von Ståhl und Bosch [SB14] beschriebt, wie CI in industriellen Softwareentwicklungen eingesetzt wird und welche Unterschiede in der Umsetzung der Prinzipien in verschiedenen Projekten bestehen.

Testfallauswahl (Test Case Selection) Reduziert man die Menge der zu einem Zeitpunkt und in Bezug auf bestimmte Code-Änderungen auszuführenden Tests, so spricht man von Testfallauswahl [RH96; Gra+01]. Eine Möglichkeit Tests auszuschließen bieten Auswirkungs- bzw. Abhängigkeitsanalyse (siehe dazu auch den Punkt *Test Impact Analyse*). Wird beispielsweise in einem isolierten Teil eines Systems etwas geändert, das nachweisbar keine anderen Stellen des Programms beeinflusst, können Tests, die nur die letztgenannten anderen Stellen abdecken, ausgelassen werden.

Minimierung der Testfälle (Test Case Minimization) Als Minimierung der Testsuite oder der Testfälle wird das Entfernen redundanter Tests aus der verfügbaren Testfallmenge bezeichnet. Dies geschieht nach Feststellung der Redundanz.

Testfallpriorisierung (Test Case Prioritization) Ist eine Menge an Tests gegeben und soll eine Liste erzeugt werden, die festlegt, in welcher Reihenfolge diese Tests ausgeführt werden, so spricht man von Testfallpriorisierung. Die Priorisierung kann sowohl für automatische als auch für manuelle Tests erfolgen. Metriken, anhand derer die Tests sortiert werden können, sind beispielsweise deren letztbekannte Ausführungsdauer oder die Anzahl an insgesamt aufgerufenen Methoden. Allerdings können auch andere Faktoren in die Berechnung mit einfließen wie beispielsweise die Häufigkeit, mit der ein Test fehlschlägt. [Rot+01; Jür+11]

Test Impact Analyse (TIA) Bei der TIA werden Informationen, die während der Testausführung aufgezeichnet wurden, mit Programmteilen verknüpft, die vom Test ausgeführt wurden. Diese Relation gilt auch in die umgekehrte Richtung. Findet eine Änderung an Programmteilen statt, die von einem Test abgedeckt wurden, so kann sich diese Änderung auf das Testergebnis auswirken – der Test gilt als *impacted*. Die TIA dient in

Teamscale (siehe weiter unten) der Implementation des in der Forschungsgruppe vorgeschlagen hybriden Testselektions- und Priorisierungsverfahrens.

Change Driven Testing Wird der Testprozess auf aufgetretene Change Sets ausgerichtet, so spricht man von Change Driven Testing. Ein mittlerweile in der Praxis eingesetztes Verfahren stellt die Ticket-Coverage [Rot+17] dar, die überprüft, wie vollständig die Änderungen sind, die in einem bestimmten Ticket (neues Feature, Änderungswunsch etc.) getestet wurden.

Mining Software Repositories Versionskontrollsysteme, Issue-Tracker und Build-Systeme lassen sich nicht nur für ihren unmittelbaren Zweck verwenden. Daten, die aus ihnen extrahiert werden, können zur Untersuchung von Entwicklerverhalten oder Build-Aktivität beitragen.

So werden in dieser Arbeit Daten von Travis CI und Github verwendet, die vor allem zur Selektion der Forschungsobjekte dienen.

Mutationsanalyse Bei der Anwendung von Mutationsanalysen werden automatisch Fehler ($\hat{=}$ Mutanten) eingestreut, um herauszufinden, ob die vorhandene Testsuite diese Fehler aufdecken kann.

Mutanten sind Variationen einer Methode, in denen – zum Beispiel durch das Ersetzen eines Operators oder eines Rückgabewertes – vom Originalverhalten abgewichen wird.

Kann ein Test, der eine bestimmte Methode durchläuft, die Mutation in dieser Methode nicht erkennen, schlägt der Test also nicht fehl, so wird davon gesprochen, dass die Methode durch ihn lediglich *scheingetestet* ist.

Auswirkungsanalyse im Software Engineering Dieses Gebiet beschäftigt sich mit der Frage, welche Konsequenzen mit der Änderung an einer Stelle im Quelltext einhergehen. Im Rahmen dieser Arbeit wird besonders auf die Auswirkungen von Änderungen im Produktivcode zu vorhandenen Testfällen geachtet.

Testfallspezifische Coverage Testfall- oder testspezifische Coverage beschreibt die erhobenen Ausführungsinformationen, die einem einzelnen Testfall zuzuordnen sind. Dazu wird beim Start beziehungsweise Ende eines Testfalls das Erhebungswerkzeug mit einer entsprechenden Test-ID informiert. So kann dieses die Information einem einzelnen Testfall zuordnen und gesondert abspeichern. Im Kontext von Testpriorisierungsverfahren wird die testfallspezifische Coverage unter anderem von Dreier [Dre17] genannt. Wie generell bei der Coverageerhebung ist auch die testfallspezifische Coverageerhebung auf unterschiedlichen Granularitäten, zum Beispiel auf Zeilen- oder Methodenebene, möglich. Durch testfallspezifische Coverage kann eine Beziehung von Tests zu Codestellen im Produktivcode hergestellt werden.

In das Software-Werkzeug Teamscale (vgl. weiter unten) lässt sich testspezifische Coverage importieren und für eine Testausführung anzeigen (Abb. 2.1). Außerdem lässt sich in der Quelltext-Ansicht von einer Methode auf Tests schließen, die dieselbe ausführen (Abb. 2.2 und 2.3).

GitHub-Travis-Projekt In der hier vorliegenden Studie werden einige Projekte untersucht, deren Quelltext auf GitHub frei verfügbar ist und die Travis als CI-Umgebung verwenden. Diese Projekte werden hier „GitHub-Travis-Projekte“ genannt.

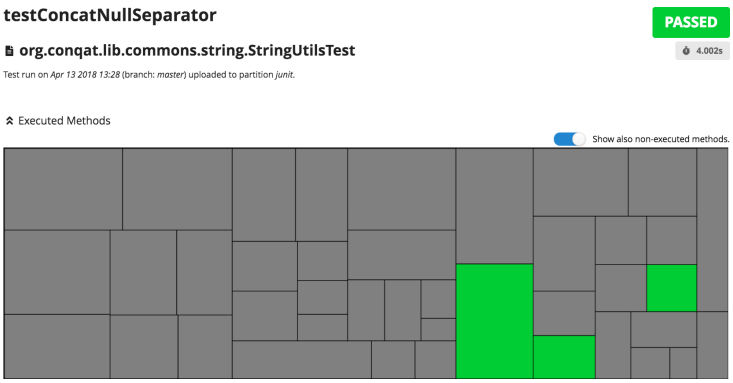


Abbildung 2.1: Die Darstellung von testspezifischer Coverage geschieht in Teamscale durch eine Treemap. Hierbei steht jedes Rechteck für eine Methode, wobei die grün markierten Rechtecke ausgeführte Methoden darstellen. Bewegt man den Mauszeiger über die Rechtecke, werden Informationen zur Methode eingeblendet.

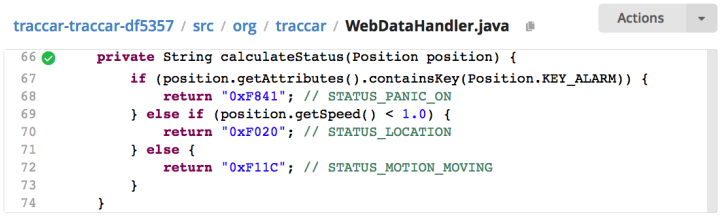


Abbildung 2.2: In Teamscale wird in der Code-Ansicht durch den grün hinterlegten Haken neben der Zeilennummer dargestellt, dass die Methode durch einen Test ausgeführt wurde. Nach einem Klick auf diesen Haken wird eine Liste von Tests angezeigt, die diese Methode ausführen (siehe Abb. 2.3).

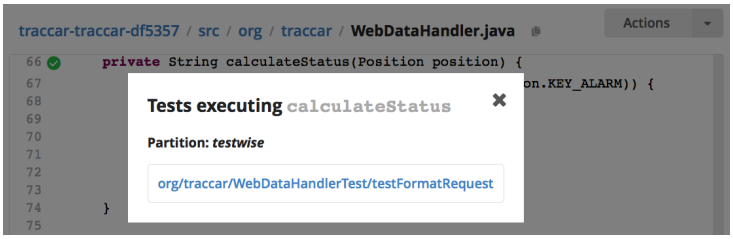


Abbildung 2.3: Wird eine Methode durch mindestens einen Test überdeckt, so kann Teamscale die Liste von Tests, die diejenige Methode ausführen, anzeigen.

Time to First Failure Die Zeit bis zum ersten Testfehlschlag (in dieser Arbeit „Time to First Failure“ oder „Time to Failure“) kann gemessen werden, um festzustellen, wie lange es in einer priorisierten Testausführung dauert, bis ein Softwaretest fehlschlägt. Dies ist eine relevante Metrik, um die Nützlichkeit eines Priorisierungsalgorithmus zu bewerten. Denn die Ausführungszeit eines Build wird maßgeblich durch die Ausführungsdauer der Tests bestimmt [SB14]. In einer Software-Entwicklungsumgebung, die den Grundsätzen von Continuous Integration (siehe oben) folgt, sind Entwickler angehalten, häufig ihren lokalen Codestand mit Veränderungen an das Versionsverwaltungssystem zu übergeben. An solch einen „Push“ ist meist die Ausführung eines Build gebunden. Dauern die Tests, also auch der Build lange, verzögert sich die Rückmeldung des Build-Systems an den Entwickler, ob durch die Änderungen Regressionstests fehlschlagen. Überschreitet die Verzögerung einige Minuten, so ist anzunehmen, dass der Entwickler eine neue Aufgabe aufgenommen hat und sich möglicherweise schon in eine andere Stelle des Programmcodes eingearbeitet hat. Kommt dann ein negatives Feedback vom Build-System (ein oder mehrere Tests schlagen fehl), muss erneut ein Kontextwechsel erfolgen, um den Fehler zu beheben.

Änderungsbasierter Regressionstest Die Wartung eines langlebigen Softwaresystems macht innerhalb dessen Lebenszeit den größten Aufwand aus. Basil Sherlund stellte 1995 auf der Konferenz „Defect Prevention – 12th International Conference and Exposition on Testing Computer Software“ in seiner Präsentation [She95] seinen Ansatz vor, der (wie die hier vorliegende Arbeit) den Fokus darauf legt, Modifikationen im Programm zu testen. Ein in der Forschungsgruppe vorgeschlagener, kombinierter Selektions- und Priorisierungsalgorithmus verwendet ein ähnliches Prinzip als Grundlage. Durch die Nutzung vorher aufgenommener testspezifischer Daten kann der Testprozess so optimiert werden, dass Änderungen früh durch einen Test abgedeckt werden.

Teamscale Teamscale[1] ist ein Softwareprodukt der CQSE GmbH (Firmensitz: München) und dient der kontinuierlichen Qualitätsanalyse von in Entwicklung befindlicher Software. Teamscale hilft bei der Detektion von Quelltextduplikaten, kann Architekturkonformitätsanalysen durchführen, untersucht inkrementell die von den Entwicklern getätigten Commits, warnt bei potentiell problematischen Stellen im Quelltext und kann Aspekte der Softwarequalität unterschiedlicher Revisionen miteinander vergleichen. Außerdem – dies ist für das Thema dieser Arbeit besonders relevant – unterstützt Teamscale den Import von testspezifischer Coverage und ist mit einer Testpriorisierungs- und Selektionslogik ausgestattet.

Teamscale wird im Rahmen der Studie als das zugrundeliegende Analysewerkzeug verwendet. Um den spezifischen Anforderungen dieser Studie gerecht zu werden, wurde das Werkzeug an einigen Stellen angepasst und erweitert.

1 `https://www.teamscale.com`

3 Verwandte Arbeiten in der Literatur und der Forschungsgruppe

In der Softwareentwicklung spielen Softwaretests eine wichtige Rolle. Wächst die Menge der Tests und damit auch die Ausführungszeit der gesamten Suite, so können Tests möglicherweise nicht mehr so häufig ausgeführt werden, wie es wünschenswert ist, oder die langen Rückmeldezeiten verhindern eine effiziente Arbeitsweise der Entwickler.

Aus diesem Grund werden Möglichkeiten, die Anzahl an auszuführenden Tests zu reduzieren oder zu priorisieren, stetig erforscht und die entsprechenden Ergebnisse veröffentlicht.

In diesem Kapitel soll knapp auf verwandte Arbeiten in der Literatur, auf frühere Arbeiten in der Forschungsgruppe und auf parallel laufende Arbeiten eingegangen werden.

3.1 Veröffentlichungen in der Literatur

Es ist wahrlich spannend zu sehen, wie früh die Forschung zu Testselektion und -priorisierung begonnen hat und wie aktuell dieses Thema auch in der heutigen Zeit ist.

Fischer et al. beschreiben schon im Jahr 1981 das Problem, dass die „Retest all"-Strategie (also die erneute Ausführung aller verfügbaren Tests) sehr teuer ist und die Möglichkeiten der Testselektion sich vor allem auf das Wissen einzelner Personen beschränken [FRC81]. Zwar haben sich die Umstände seit dieser Zeit durch die massiv fortgeschrittene Technik und heute meist automatisiert ausgeführte Tests selbstredend verändert, doch bleibt das Ziel im Grunde das gleiche – nämlich durch Regressionstests die durch Modifikation am Programm möglicherweise entstandenden Fehler früh aufzudecken. Weitere Artikel beschäftigten sich mit diesem Problem bereits in den 1990er Jahren: 1994 wird „TestTube" als ein System für selektives Regressionstesten vorgestellt [CRV94], das statische und dynamische Codeanalysen nutzt, und auch Sherlund stellte in einer Präsentation 1995 ein Testauswahlsystem vor [She95]. Hsia et al. beschäftigten sich 1997 mit den Schwierigkeiten, die durch objektorientierte Sprachen und damit erschwerte Abhängigkeitsanlysen auftraten [Hsi+97]. Die Schwierigkeit, an für die Priorisierung relevante Daten zu kommen, hat sich seit der Veröffentlichungszeit dieser älteren Literatur verringert. Nicht verändert hat sich hingegen der logische Hintergrund, den Fischer et al. wie folgt formulieren:

- „For a given modification, what other section(s) of the software is impacted by that modification?"
- „For the identified section(s) of code that could be affected, what test cases should be rerun to assure the proper execution of existing capability?" [FRC81, S. 1]

Nutzt man die Test Impact Analyse, so werden Antworten auf diese Fragen aus Daten vergangener Testausführungen berechnet. Testspezifische Coveragedaten erlauben es, von einem Test auf die durch ihn ausgeführten Methoden zu schließen. Findet später in einer Methode, die durch einen Test abgedeckt wird, eine Änderung statt, soll der abdeckende Test erneut ausgeführt werden.

Literatur von heute bezieht sich auf moderne Techniken, um intelligenter und damit effizienter zu testen. Vorschläge für neue Priorisierungs- oder Selektionsverfahren finden

sich häufig. Ein aktuelles Beispiel hierfür ist ein innerhalb der Facebook Inc. getestetes Verfahren, das maschinelles Lernen einsetzt [Mac+18]. Eine Übersichtsstudie über Verfahren zur Testsuiteminimierung, Testfallselektion und -prioriserung aus dem Jahr 2012 bieten Yoo et al. in [YH12].

Häufig werden neu vorgeschlagene Priorisierungs- oder Selektionsverfahren anhand von Mutationsanalysen im Zuge der Vorstellung evaluiert [Lu+16]. Dabei wird überprüft, ob und wie schnell das Verfahren einen eingebauten Mutanten aufdecken kann. Seltener werden in der Literatur Fälle beschrieben, in denen Codeänderungen tatsächlich händisch eingepflegt und darauf Priorisierungsverfahren angewandt werden [Lu+16; Rot+99; Elb+04]. So modifizieren Lu et al. in ihrer Studie [Lu+16] Quelltextabschnitte, fügen neue Tests hinzu und untersuchen die Auswirkungen ihres Tuns auf Priorisierungsverfahren.

3.2 Frühere Arbeiten in der Forschungsgruppe

Einen Grundstein der Test Impact Forschung innerhalb der Forschungsgruppe legte die Masterarbeit von Florian Dreier. In ihr wurde ein Tooling zur testfallspezifischen Coverageerhebung implementiert und unter Nutzung der damit erhobenen Daten ein kombiniertes Testselektions und -priorisierungsverfahren entwickelt [Dre17].

In einer weiteren Masterarbeit wurde dieses Verfahren in einem Automobilkonzern, der eingebettete Systeme entwickelt, getestet und evaluiert [Ein18].

Eine weitere Forschungsarbeit [Rot18] lässt sich als die Vorgängerarbeit zur vorliegenden Arbeit bezeichnen. In ihr wurde der Prototyp für ein Benchmarksystem für unterschiedliche Testpriorisierungsverfahren entworfen und anhand mehrerer Open Source Projekte getestet.

3.3 Parallel laufende Arbeiten

Zugleich mit der Erstellung der hier vorliegenden Arbeit erforscht Rainer Niedermayr (Doktorand an der Universität Stuttgart und Mitarbeiter der CQSE GmbH), wie sich sich die Effektivität von Tests besser messen lassen kann. Da die Effektivität von Tests in der Priorisierung selbiger eine wesentliche Rolle spielt, finden sich im Laufe dieser Arbeit immer wieder Bezüge auf die Arbeit von Niedermayr.

Manche Methoden eines Systems lassen sich beispielsweise als „trivial“ klassifizieren. Diese sind möglicherweise nicht in gleichem Maße testenswert wie andere Methoden des Systems und können deshalb im Priorisierungsprozess anders gehandhabt werden [NRW18b]. Eine weitere Möglichkeit, die Niedermayr untersucht, ist die Einbringung von Daten aus Mutationsanalysen in die Testpriorisierung.

Derzeit wird an der Technischen Universität München auch eine industrienahe Masterarbeit bearbeitet, die die Einsetzbarkeit der Test Impact Analyse in der Entwicklung von betrieblichen Informationssystemen mit großen Testsuiten evaluiert.

4 Analyse und Anpassung der bestehenden Test-Impact-Analyse

> *The purist will demand that all previously used test cases be rerun. The pragmatist will leave the decision to the discretion of the test director as he believes the test director knows the software best, and by using engineering judgement and his knowledge of the code he often manually selects the subset of previously completed test cases to be rerun.*
>
> FISCHER ET AL., 1981 [FRC81]

Durch Arbeiten in der Forschungsgruppe und durch darauf aufbauende Softwareentwicklungen von der CQSE GmbH ist eine Implementierung der Test-Impact-Analyse entstanden. In diesem Kapitel wird der Stand dieser Implementation, wie er zu Beginn der Arbeit auf einem Forschungsstrang im Git-Repository von Teamscale verfügbar war, grob beschrieben. Da die Analyse Gegenstand aktueller Entwicklung ist, konnten häufige Änderungen an der Funktionalität und in der Handhabung der Werkzeuge zur Testpriorisierung beobachtet werden.

Die Test-Impact-Analyse lässt sowohl eine Priorisierung als auch eine Selektion von Testfällen zu. Ein automatisiertes Testsystem muss sich also nicht, wie Fischer et al. (siehe Eingangszitat oben) beschrieben haben, auf die Aussagen von Testmanagern stützen, sondern kann Daten aus vorausgegangenen Testausführungen nutzen.

4.1 Beschreibung der bestehenden Umsetzung

Test-Impact-Analyse zur Erstellung von geordneten Testausführungslisten arbeitet auf zwei Datenquellen: zum einen auf einem Programmquelltext-Delta (Unterschied) von einer Version zu einer nächsten, die getestet werden soll, und zum anderen auf Informationen, die während der Ausführung der Test zur ersten Version aufgezeichnet wurden.

In Teamscale ist dieses System implementiert. Durch die Verknüpfung mit einem Versionskontrollsystem sind Quelltext und Änderungen am selben bekannt. Testausführungsinformationen können mithilfe eines frei verfügbaren Java-Agents aufgezeichnet und im Anschluss an Teamscale hochgeladen werden. In Teamscale werden Code und Ausführungsinformationen miteinander verknüpft. Für die TIA findet dies in der aktuellen Version auf Methodenebene statt. Durch die Verwendung von testspezifischer Coverage ist bekannt welcher Test welche Methoden ausgeführt hat. Umgekehrt kann nach der Änderung an einer Methode auf die Menge von Tests geschlossen werden, die die Methode in früheren Testdurchläufen ausgeführt haben.

Teamscale überprüft regelmäßig, ob dem Versionskontrollsystem des zu analysierenden Projekts eine neue Quelltext-Version übergeben wurde und registriert die Änderungen. Auf Basis der in Teamscale gespeicherten Daten kann eine Testausführungsreihenfolge für die neue Quelltext-Version errechnet werden. Um eine solche Testreihenfolge im Buildprozess zu berücksichtigen, ist eine Unterstützung zum Beispiel für JUnit geplant, sodass vor der Ausführung des Tests der Test-Runner den entsprechenden Teamscale-Dienst aufruft und die vorgeschlagene sortierte Testliste ausführt.

4.2 Untersuchung des bestehenden Systems

In der Arbeit von Rott [Rot18] wurden verschiedene Priorisierungalgorithmen evaluiert, wobei die im letzten Abschnitt vorgestellte Test-Impact-Analyse zum Einsatz kam.

Im Rahmen dieser Arbeit ergaben sich Auffälligkeiten, welche untersucht wurden. Die nachfolgenden Unterabschnitte gehen auf die gewonnenen Erkenntnisse ein. Es wird über die Untersuchung der erhobenen Testausführungszeiten berichtet, die sich von den vermeintlichen Ausführungszeiten im JUnit-Test-Report unterschieden. Weiterhin wurden 15 Builds untersucht, in denen die Testselektion den fehlschlagenden Test nicht zur Ausführung vorgeschlagen hatte.

Untersuchung der von Rott aufgenommenen Ausführungszeiten von Tests

Die Aufzeichnung der Coverage und Ausführungszeiten wurde in der Arbeit von Rott [Rot18] mit dem Werkzeug `coverage-conductor` durchgeführt. Obwohl zu einem frühen Zeitpunkt der Studie stichprobenartig durch Vergleich mit den im JUnit-Bericht angegebenen Zeiten kontrolliert wurde, ob die während der Testausführung erhobenen Daten plausibel sind, ließen sich für einige Tests Abweichungen der mit dem `coverage-conductor` aufgezeichneten Testzeitspannen von den Zeiten im JUnit-Bericht feststellen. Da die Ausführungszeiten der Tests für die Priorisierung verwendet werden, zieht eine Abweichung hier eine geänderte Testreihenfolge nach sich. Die Ursache für die abweichenden Zeiten konnte in [Rot18] nicht ausgemacht werden.

Im Rahmen der hier vorliegenden Arbeit wurde untersucht, ob die mit dem `coverage-conductor` aufgezeichneten Zeiten korrekt sind. Dafür wurden in einem Projekt (spring-data-examples, Commit-Stand: c523b8) in zufällig ausgewählten Methoden nacheinander Ausführungsverzögerungen `Thread.sleep(..)` hinzugefügt. Hierbei lieferte der `coverage-conductor` die erwarteten Werte; die Zeiten im JUnit-Report wichen ab und zeigten statt der erwarteten Dauer ein Sekunden-Offset vom Testausführungsstart einer Testklasse bis zum Abschluss eines Testfalls. In einem weiteren Projekt konnte wiederum keine Abweichung festgestellt werden.

Durch die künstliche Verlängerung einer Methode wurde herausgefunden, dass die mit dem verwendeten Tool aufgenommenen Daten korrekt sind und die Daten im JUnit-Report fälschlicherweise davon abweichen. Die im JUnit-Report über alle Tests aggregierte Testausführungsdauer von 32 Minuten wich außerdem von der realen Ausführungsdauer von etwa 8 Minuten stark ab.

Aufgrund der fortgeschrittenen Entwicklung von Teamscale und dazugehöriger Plugins erfolgt in der hier vorliegenden Studie die Zeitnahme nicht über den `coverage-conductor`, sondern über den Teamscale Jacoco Agent.

Nichtaufdeckung fehlgeschlagener Tests durch das in der Forschungsgruppe vorgeschlagene Selektionsverfahren in [Rot18]

Bei der Untersuchung von insgesamt 15 Builds, in denen bei der Anwendung des Selektionsalgorithmus der fehlschlagende Test nicht in der Liste der auszuführenden Tests lag, konnten fünf verschiedene Kategorien von möglichen Gründen identifiziert werden. Bis auf den Fall eines falsch abgelegten Namens eines fehlschlagenden Tests können alle Fälle im weiteren Sinne zu flackernden Tests gezählt werden. Die Befunde sind in den folgenden zwei Absätzen beschrieben.

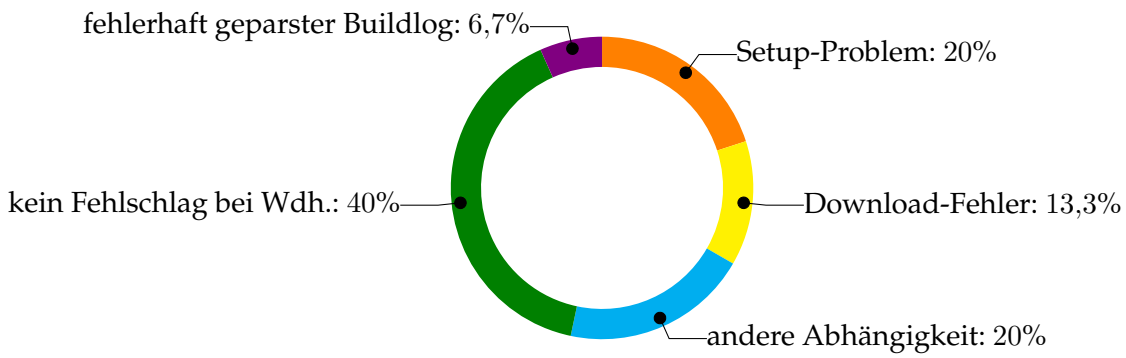


Abbildung 4.1: Von den untersuchten Builds, in denen der laut Travis fehlschlagende Test nicht ausgeführt wurde, waren die meisten flackernde Tests; dazu sind auch die Tests zu zählen, die wegen eines gescheiterten Downloads oder Abhängigkeiten vom Zustand des System unter Tests fehlschlagen.

Flackernde Tests Im Projekt Biojava gab es in zwei von drei angesehenen Builds während des Testprozesses Fehler beim *Download* einer Ressource, weswegen der Test fehlschlug. Eine gleichzeitige Änderung an der ausführenden Methode fand nicht statt.

In den Projekten jackson-databind, java-design-patterns und spring-data-examples schlug je ein Test wegen einer nicht näher untersuchten Abhängigkeit fehl.

Zwei Male schlug im Projekt jmeter-plugins ein Test wegen eines `NoClassDefFoundError` fehl. Diese Builds sowie ein weiterer Build des Projekts graphhopper scheiterten an einem Setup, das zeitweise zu Fehlschlägen führt.

In sechs Fällen schlug der laut Travis CI fehlschlagende Test bei einer erneuten Ausführung auf einem lokalen Computer nicht fehl. Dies war der Fall bei je einem Build der Projekte java-design-patterns, biojava, graylog und dropwizard sowie bei zwei Builds des Projekts graphhopper.

Falschinformation im Benchmarksystem Im untersuchten Build des Projektes logback wurde festgestellt, dass der fehlschlagende Test, der durch die Analyse der Buildlogs mithilfe des Werkzeugs `travistorrent-tools` identifiziert wurde, vom Buildlog-Analyser nicht richtig erkannt wurde. Der angeblich gescheiterte Test hatte ein positives Testergebnis, wohingehen ein anderer Test fehlschlug. Dieser andere Test fand sich in der Liste der auszuführenden Tests, die der benutzte Selektionsalgorithmus berechnete.

Vergleich der Menge fehlschlagender Tests bei Travis mit der Menge bei erneuter Ausführung

In der Vorgängerarbeit [Rot18] wurden als zu untersuchende Builds solche gewählt, die auf Travis CI fehlschlugen. Diese Builds wurden während der vorliegenden Studie nicht erneut ausgeführt, und es ist unklar, ob Testfälle, die beim Travis CI Build fehlschlugen, auch bei erneuter Ausführung reproduzierbar fehlschlagen. 435 der 437 Builds, die als Datengrundlage dienten, wurden erneut ausgeführt und die Mengen der fehlschlagenden Tests verglichen. In 103 von 435 Fällen schlugen alle Tests, die auf Travis CI fehlschlugen, reproduzierbar auch bei erneuter Testausführung fehl. In sechs Fällen schlugen bei der wiederholten Ausführung exakt die gleichen Tests und keine weiteren fehl, wie zuvor auf der Travis-Infrastruktur.

4.3 Anpassungen und Neuimplementierungen

Für die vorliegende Arbeit wurden maßgebliche Änderungen am bestehenden TIA-System vorgenommen.
Diese fanden zum einen an Teamscale statt, das die TIA-Ergebnisse verarbeitet, und zum anderen am Teamscale JaCoCo Agent, der den oben erwähnten `coverage-conductor` ablöst und als Javaagent die Erhebung von Daten während der Testausführung übernimmt.

4.3.1 Aufnahme von @BeforeClass- und @AfterClass-Methoden

Der `coverage-conductor` erhob Daten nur vom Beginn eines einzelnen Testfalls bis zum Ende des Testfalls. Testausführungsinformationen, die möglicherweise in der Zwischenzeit aufgenommen worden waren, wurden verworfen.
Zwischenzeitlich wurde der `coverage-conductor` durch den Teamscale Jacoco Agent ersetzt. In einer für diese Arbeit implementierten Version, werden Ausführungsinformationen von @BeforeClass- und @AfterClass-Methoden aufgezeichnet und in der Berechnung der geordneten Testausführungsliste entsprechend berücksichtigt.

4.3.2 Aufnahme der Modulzugehörigkeit von Tests

Projekte bestehen oft nicht nur aus einem einzelnen, sondern aus mehreren Modulen. Ohne eine spezielle Projektkonfiguration werden diese Module meist sequentiell ausgeführt. Es kann deshalb wichtig sein, die Modulzugehörigkeit von Tests bei der Priorisierung zu berücksichtigen, um nur tatsächlich ausführbare Testreihenfolgen vorzuschlagen.
Frühere Versionen des Testausführungsinformation-Rekorders und der TIA innerhalb Teamscale betrachteten die Modulzugehörigkeit von Tests nicht. Dies wurde nun im Java-Agent und auf der teamscaleseitigen Implementierung ergänzt.

4.3.3 HTTP Kommunikation des Test Listeners mit dem Javaagent

Der verwendete Java Agent lauscht an einem Port des Hostsystems und wartet auf Eingaben, die Ereignisse (Events) der Testausführung beschreiben. Dabei handelt es sich beispielsweise um Meldungen, die den Start einer Testsuite oder eines Testfalls und deren Beendigung betreffen.
Es wurde ein Test Listener entwickelt, der wenige Teile aus dem genutzten Listener in [Dre17; Rot18] übernimmt und nun zur HTTP Kommunikation mit dem Temascale Jacoco Agent fähig ist. Normalerweise wird dem Teamscale Jacoco Agent eine zu nutzende Portnummer übergeben. Um eine parallele Testausführung und -aufnahme zu ermöglichen, wurden Listener und Javaagent um ein zufälliges Portauswahlsystem ergänzt. Dabei erfolgt die Informationsweitergabe über eine automatisch vom Agent angelegte Textdatei in einem spezifischen Verzeichnis des Dateisystems.

4.3.4 Unterstützung Surefire- und Failsafe-Plugin

In der früheren Version des Benchmarking-Systems [Rot18] war die Aufzeichnung von Testcoverage auf Tests beschränkt, deren Ausführung durch das Maven Surefire-Plugin stattfand. Integrationtests werden allerdings häufig durch das Maven Failsafe-Plugin ausgeführt. Um möglichst viele Buildkonfigurationen abzudecken, wurde die Coverageaufnahme auf das Maven Failesafe-Plugin erweitert.

4.3.5 Migration der Erweiterung der TIA auf eine neuere Version von Teamscale

Gemeinsam mit einem Mitarbeiter der CQSE wurden zwei Entwicklungsstränge des Teamscale-Repositorys, die für Forschungsarbeiten angelegt worden waren und auf einer älteren Version von Teamscale aufsetzten, zusammengeführt. Zwischenzeitlich wurde das System zur Integration von testspezifischer Coverage geänderte und effizienter gemacht. Zusätzlich kann so zum einen von anderen, neuen Funktionen von Teamscale (wie beispielsweise der Darstellung testspezifischer Coverage in einer Treemap, siehe Abb. 2.1) Gebrauch gemacht werden.

Eine bedeutende Änderung, die im Zuge der Aktualisierung übernommen wurde, stellt die Trennung von Code und Testdaten dar. Zum Zeitpunkt der Vorgängerarbeit [Rot18] waren Informationen zu einer Testausführung mit der entsprechenden Quelltextdatei (eines Tests) verknüpft. In dieser älteren Version wurde die Menge an Tests, die in einem Projekt existieren, aus entsprechenden Java-Annotationen (`@Test`) extrahiert. In der aktuellen Version ist es nun möglich, dass kein Quelltext von Tests der Analyse in Teamscale zugeführt wird. Testresultate (zum Beispiel PASSED, FAILED) und deren Ausführungszeiten können unabhängig von in Teamscale integriertem Quellcode in das Analysewerkzeug integriert werden. Testspezifische Coverage, die Produktivcode abdeckt, wird weiterhin mit den entsprechenden Quelltextstellen verknüpft. Durch die Trennung von Quellcode und Testdaten ist es nicht mehr möglich, durch das Parsen von Methodenannotationen neue oder gelöschte Tests zu detektieren. Ein entsprechendes Verfahren zur Handhabung solcher Tests wurde von der CQSE GmbH implementiert, allerdings erst im späteren Verlauf der hier vorliegenden Arbeit. Dabei werden beim Abruf des Teamscale-Dienstes zur Berechnung einer priorisierten Testliste dem Aufruf die aktuell verfügbaren Tests angehängt. Diese Methode wurde nicht in das Forschungssetup übernommen, sondern es wurde eine eigene Funktionalität implementiert, die die Tests zum ersten und zweiten Codestand über eine Datenbank abgleicht.

4.3.6 Verbesserungen hinsichtlich parametrisierter Tests

Die aktualisierte Version der TIA Implementierung ermöglicht es nun, parametrisierte Tests in der Berechnung der Testausführungsreihenfolge gesondert zu betrachten.

Standardmäßig werden parametrisierte Tests nicht mehr in beliebiger Reihenfolge vorgeschlagen, sondern vor der Priorisierung zusammengefasst und als einzelner Test behandelt, der auch nur als eine Einheit ausgeführt werden kann.

5 Forschungsfragen

Im Rahmen dieser Arbeit wurde ein System teils neu-, teils weiterentwickelt, mit dem die Zeit bis zum ersten auftretenden Testfehlschlag gemessen werden kann. Diese Messung geschieht auf Basis einer Liste von selektierten und priorisierten Testfällen. Dabei werden einerseits Daten aus in der Vergangenheit ausgeführten Builds verwendet und andererseits Versionen eines zu untersuchenden Systems mit eingestreuten Mutationen.

Die nachstehenden Forschungsfragen sollen mithilfe dieses Systems beantwortet werden. Die Antworten darauf tragen zur der Bewertung des an der Forschungsgruppe entwickelten Selektions- und Priorisierungsverfahrens bei. Die Ergebnisse aus der Performanzanalyse auf Basis bereits ausgeführter Builds werden in der Diskussion den Ergebnissen aus Mutationsdaten vergleichend gegenübergestellt.

Die Realitätsoptionen, die in das Benchmarksystem implementiert wurden, ermöglichen es auch, unterschiedliche – sowohl heute existierende, als auch künftig denkbare – Funktionalitäten eines Testsystems nachzustellen.

Die Fragen werden im Folgenden einzeln besprochen und eine grobe Vorgehensweise zu ihrer Bearbeitung beschrieben.

5.1 Fragestellungen anhand historischer Buildfehlschläge

Zunächst soll die Effizienz von Priorisierungsverfahren an GitHub-Travis-Projekten evaluiert werden.

Die in Abschnitt 8.3.1 vorgestellten Realitätsoptionen sind in den folgenden sechs Fragen RQ 1.1–1.3 und RQ 2.1–2.3 auf die Standardvariante festgelegt:

- Testabhängigkeiten (@AfterClass- und @BeforeClass-Methoden) werden behandelt.
- Tests werden dem Modul ihrer Zugehörigkeit nach geclustert.
- Parametrisierte Tests werden als nur gemeinsam ausführbar beahndelt.
- Testfälle einer Klasse müssen in der Testausführungsliste nicht direkt aufeinander folgen.

5.1.1 RQ 1.1: Welcher Algorithmus führt durchschnittlich zur kürzesten Time to Failure?

Eine geringe Dauer bis zum ersten Testfehlschlag hilft, im Entwicklungsprozess einen Fehler früh aufzudecken und früh zu beheben. Eine effiziente Strategie zur Ausführung von Tests soll deshalb die Testausführungsreihenfolge so beeinflussen, dass der erste Testfehlschlag möglichst früh erreicht wird.

Aufgrund einer vorhergehenden Analyse von Buildlogs, sind für ausgewählte Projekte Testfehler bekannt, die zu einem bestimmten Codestand auftraten (roter Build oder roter Commit). Durch die Ausführung verschiedener Strategien zur Bildung einer geeigneten Testausführungsliste, soll festgestellt werden, durch welches Verfahren dieser Testfehlschlag am schnellsten zu erreichen ist (Time to Failure). Um geänderte Codestellen zu ermitteln, wird der ausgewählte Codestand mit einem Vorgänger (grüner Vorgänger-Build

oder grüner Commit) verglichen und Selektions- sowie Priorisierungsalgorithmen sollen Testausführungsinformationen aus der Testausführung des Vorgängers verwenden können.

5.1.2 RQ 1.2: Wie hoch ist der Anteil an impacted Tests vom letzten erfolgreichen bis zum fehlschlagenden Build?

Der von der Forschungsgruppe vorgeschlagene Algorithmus bietet verschiedene Möglichkeiten, Tests von der Ausführung auszuschließen. Innerhalb dieser Forschungsfrage soll geklärt werden, wie hoch jeweils der Anteil an Testfällen ist, die durch die Selektion ausgewählt wurde.

5.1.3 RQ 1.3: Häufigkeit von unentdeckten Testfehlschlägen durch die Selektierung von Testfällen mit dem vorgeschlagenen Verfahren und optionalen Anpassungen

Der in der Forschungsgruppe vorgeschlagene Selektionsmechanismus nutzt die Daten des Code-Deltas von einer eingecheckten Version des Quelltextes zu einer anderen. In der Realität schlagen Tests allerdings nicht immer aufgrund einer Codeänderung fehl.

Die Forschungsfrage RQ 1.3 soll deshlab klären, wie viele auf Travis fehlgeschlagene Tests durch die Anwendung der Selektion nicht aufgedeckt werden können.

5.2 Fragestellungen anhand von Mutationsanalysen

Ein Ziel dieser Studie ist es, zu klären, wie stark sich die Ergebnisse vom Benchmark, der Testpriorisierungsverfahren anhand realer – in der Vergangenheit aufgetretener – Testfehlschläge evaluiert, und die Ergebnisse von Mutationsanalysen ähneln.

Da für einen Entwickler die Testdauer von wesentlicher Relevanz ist, um eingebaute Fehler schnell zu beheben, kann eine Evaluation allein anhand von Mutationsdaten unzureichend sein.

Die Ergebnisse aus den folgenden Forschungsfragen und der Vergleich mit den Ergebnissen aus den Forschungsfragen RQ 1.x sollen helfen, die Vergleichbarkeit zu hinterfragen.

5.2.1 RQ 2.1: Welcher Algorithmus führt durchschnittlich zur kürzesten Time to Failure?

Ein Testpriorierungsverfahren dient dazu, einen Test früh auszuführen, wenn die Wahrscheinlichkeit, dass selbiger einen Fehler aufdeckt, hoch ist. In dieser Forschungsfrage soll getestet werden, wann in einer priorisierten Testausführung der erzeugte Mutant durch einen fehlschlagenden Test aufgedeckt wird.

Das Code-Delta (Changeset), das der Berechnung der geordneten Testausführungsliste als Eingabe dient, soll neben der mutierten Methode auch noch fünf beziehungsweise – in einer weiteren zu berechnenden Konfiguration – zehn Methoden enthalten. So arbeitet der Dienst zur Erzeugung der Testausführungsliste mit einem realistischeren Changeset, denn real auftretende Code-Deltas sind meist auch größer als eine Methode.

5.2.2 RQ 2.2: Wie hoch ist der Anteil an impacted Tests durch die Mutationen?

Analog zur Forschungsfrage RQ 1.2 soll auch für den Mutationsbenchmark überprüft werden, wie hoch der Anteil der zur Ausführung ausgewählten Tests im Vergleich zur Gesamtmenge der Tests ist.

5.2.3 RQ 2.3: Häufigkeit von unentdeckten Testfehlschlägen durch die Selektierung von Testfällen mit dem an der Forschungsgruppe vorgeschlagenen Verfahren

Es soll geprüft werden, wie viele Fälle auftreten, in denen durch die Ausführung der Gesamtmenge der Tests ein Mutant aufgedeckt werden kann, durch die Menge der vom Selektionsalgorithmus ausgewählten Tests aber nicht.

5.3 Fragestellungen zu technischen Einschränkungen im Testverfahren und möglicher Vereinfachung im Prozess

Die Berechnung der Testausführungszeiten im Benchmark geschieht auf Basis der in Teamscale verfügbaren Daten, ohne die Tests erneut in der vorgeschlagenen Reihenfolge auszuführen. Diese Daten wurden in einer vorhergehenden Ausführung erhoben, wofür das Testsystem entsprechend instrumentiert sein muss.

Um die theoretisch berechnete Ausführungsdauer möglichst auch in Wirklichkeit erreichen zu können, sollen sich technische Einschränkungen abbilden lassen. Dies ist durch die entworfenen Realitätsoptionen möglich.

Forschungsfrage RQ 3.1 soll anhand des an der Forschungsgruppe vorgestellten Priorisierungsalgorithmus die Auswirkunngen der Realitätsoptionen untersuchen. In der darauf folgenden Forschungsfrage soll Häufigkeit und Dauer von @BeforeClass- und @AfterClass-Methoden in den untersuchten Projekten gemessen werden.

Durch Beantwortung der letzten Forschungsfrage RQ 3.3, soll festgestellt werden, ob die Performanz des in der Forschungsgruppe vorgestellen Algorithmus abnimmt, wenn ältere Testausführungsdaten zur Priorisierung verwendet werden.

5.3.1 RQ 3.1: Wie wirken sich Beschränkungen in der Ausführungsreihenfolge von Tests (z. B. innerhalb von Modulen oder Klassen) auf die Time to Failure aus?

Die neu implementierten Funktionalitäten (siehe Kapitel 4)

- zur erweiterten Handhabung (Zusammenführung) parametrisierter Tests,
- der modulweisen Behandlung von Tests,
- der Möglichkeit abzubilden, dass alle Tests einer Klasse hintereinander ausgeführt werden müssen und der Beachtung von Testinformationen hinsichtlich @BeforeClass- und @AfterClass-Methoden,

erlauben es, den Benchmark in unterschiedlichen Konfigurationen auszuführen.

Damit lassen sich technische Einschränkungen eines Testsystems (zum Beispiel Maven Surefire) nachstellen, und die Benchmarkergebnisse bilden real erreichbare Testzeiten besser ab.

Die Einstellungen hinsichtlich der oben genannten vier Optionen werden – wie in Abschnitt 8.3.1 – Realitätsoptionen beziehungsweise Realitätskonfiguration genannt.

Zusätzlich erlaubt es ein Vergleich zu sehen, an welchen Stellen eines Testsystems es sich möglicherweise lohnt, die Funktionalität zu erweitern.

In dieser Forschungsfrage sollen die Time to Failure-Werte für den Selektions- und Priorisierungsalgorithmus der Forschungsgruppe unter Verwendung der nachfolgend genannten vier Realitätskonfigurationen ausgeführt werden:

- **CMDP**: Alle Tests einer Klasse müssen hintereinander ausgeführt werden, Tests werden modulweise ausgeführt, @BeforeClass- und @AfterClass-Methoden werden nicht ausgeschlossen, parametrisierte Tests können nicht einzeln ausgeführt werden.
- **MD**: Tests werden modulweise ausgeführt, @BeforeClass- und @AfterClass-Methoden werden nicht ausgeschlossen, parametrisierte Tests können nicht einzeln ausgeführt werden.
- **MP**: Tests werden modulweise ausgeführt, @BeforeClass- und @AfterClass-Methoden werden ausgeschlossen, parametrisierte Tests können nicht einzeln ausgeführt werden.
- **w/o res**: Die Selektion und Priorisierung findet lediglich auf Basis einzelner Tests und ohne Beachtung von Modul- oder Klassengrenzen statt; parametrisierte Tests können mit einzelnen Parametern aufgerufen werden.

5.3.2 RQ 3.2: Wie häufig und langandauernd sind klasseneigene Setup- und Teardown-Methoden?

Die in dieser Studie verwendete Version des Teamscale Jacoco Agent nimmt nicht nur Testausführungsinformationen auf, die von der unmittelbaren Ausführung eines einzelnen Testfalles (Testmethode oder parametrisierter Test mit bestimmten Eingabedaten) stammen, sondern zeichnet auch in der Zeit vor und nach einem Testfall auf. Dabei handelt es sich um die @BeforeClass- und @AfterClass-Methoden, die in einer Testklasse vorhanden sein können.

In dieser Forschungsfrage soll geklärt werden, wie häufig solche Methoden in den untersuchten Projekten aufgezeichnet werden und wie lange deren Ausführung dauert.

5.3.3 RQ 3.3: Was sind die Auswirkungen einer unregelmäßigen Aktualisierung von Test-zu-Methoden-Beziehungen?

Eine vollständig ressourcenneutrale Aufzeichnung von Coverage und Dauer während der Testausführung ist nicht möglich. Lässt das Testsystem aufgrund von großen Overheads durch die Instrumentierung und Aufnahme nur eine gelegentliche Aufzeichnung dieser Informationen zu, kann es praktikabel sein, auch länger zurückliegende Testausführungsinformationen zur Priorisierung zu verwenden.

In dieser Forschungsfrage soll anhand eines Projekts geklärt werden, welche Unterschiede es in der Performanz des an der Forschungsgruppe vorgeschlagenen Selektions- und Priorisierungsverfahrens gibt, wenn die Daten des ältesten im Benchmark verwendeten Builds zur Berechnung aller weiteren Testausführungslisten verwendet werden.

6 Datenerhebung

In diesem Kapitel wird beschrieben, wie verschiedene Daten für den Benchmark erhoben wurden. Sowohl konzeptionelle als auch technische Aspekte werden beleuchtet. Der Einfluss und das Zusammenspiel einzelner Daten im Benchmark und hinsichtlich der Forschungsfragen ist im folgenden Kapitel „Studienaufbau“ beschrieben.

6.1 GitHub und Travis CI

Als GitHub-Travis-Projekte werden in dieser Arbeit Projekte bezeichnet, die ihren Quelltext auf der GitHub Plattform hosten und die zur kontinuierlichen Integration Travis CI verwenden.

Travis CI speichert die Logdateien jedes Builds und hält sie öffentlich verfügbar. Für die Entwicklung von TravisTorrent [BGZ17b] wurden Werkzeuge[1] für den Download und zur Analyse dieser Buildlogs implementiert, von denen zwei in geringfügig modifizierten Versionen in den nachfolgend beschriebenen Schritten eingesetzt wurden.

6.1.1 Identifikation von historisch fehlgeschlagenen Builds

In dieser Arbeit soll untersucht werden, wie gut Testpriorisierungsalgorithmen in der Praxis funktionieren. Um Praxisnähe zu gewährleisten, soll der Code des Systems unter Test also nicht künstlich verändert werden, sondern die Untersuchung anhand von Codeänderungen stattfinden, wie sie tatsächlich im Verlauf eines Programms aufgetreten sind.

Einstiegspunkt hierzu waren GitHub und TravisCI. GitHub ermöglicht die freie Zugänglichkeit von Quelltexten unterschiedlicher Projekte und TravisCI hält (entsprechende Nutzung vorausgesetzt) für diese Projekte Builddaten bereit. Builds, die für die Studie von Bedeutung sind, sind Builds, die fehlschlugen. Die Durchführbarkeit der Analyse ist maßgeblich davon abhängig, ob sich die zu untersuchenden Codebasen noch kompilieren lassen. Wie auch Tufano et al. feststellten, ist dies bei weitem nicht immer möglich [Tuf+17]. Zusätzlich ist im Fall der hier vorliegenden Arbeit wichtig, ob die Tests laufen und Ergebnisse bringen, die mit den früheren Ergebnissen auf der Infrastruktur von Travis CI vergleichbar sind.

6.1.2 Identifikation fehlgeschlagener Tests

Die Identifikation der fehlschlagenden Tests in einem Build ist von zentraler Bedeutung für die vorliegende Arbeit.

Für jedes Studienprojekt wurden mit dem *travis_harvester* alle Buildlog-Dateien im Zeitraum[2] von 1. Januar 2018 bis 1. Januar 2019 heruntergeladen. Anschließend wurden diese Buildlogs mit dem Werkzeug *buildlog_analysis* analysiert und insbesondere für jeden Build die fehlgeschlagenen Testfälle extrahiert. Im Rahmen dieser Arbeit wurden mehr

1 `https://github.com/TestRoots/travistorrent-tools`

2 Für Projekte, die bereits in der Vorgängerarbeit als System unter Test dienten, waren die nötigen Daten schon in einer Datenbank verfügbar.

als 22 GB Travis Build Logs analysiert, wobei hier Buildlog Daten, die bereits in der Vorgängerarbeit [Rot18] analysiert wurden, nicht mit eingerechnet sind.

6.1.3 Identifikation des grünen Vorgänger-Builds

In dem dieser Arbeit vorangehenden Prototypen wurde für die Erhebung der testspezifischen Coverage ausgehend vom Commit, zu dem es einen fehlschlagenden Build gibt, in der Commit-Historie zurückgegangen, bis ein Commit gefunden wurde, zu dem es keinen fehlschlagenden Build gibt. Diese Regel wurde insoweit gelockert, dass es für den „grünen Vorgängercommit" lediglich einen Build geben muss, der erfolgreich verlief. Für Projekte, die Travis CI nutzen, werden in einigen Fällen für einen Commit mehrere Builds (oder Build-Jobs) gestartet, die unterschiedlich konfiguriert sind (zum Beispiel: unterschiedliche Java-Versionen). Bei einer Voruntersuchung ist aufgefallen, dass zumindest in einem Projekt der Build, der eine bestimmte Java-Version verwendet, über einen langen Zeitraum hinweg fehlschlägt. So könnte hier mit dem früher verwendeten Filter kein grüner Vorgängercommit ausgemacht werden.

6.2 Auswahl relevanter Buildfehlschläge für das Studienobjekt Teamscale

Teamscale ist für diese Studie nicht nur das zugrundliegende Analysewerkzeug, sondern auch ein Studienobjekt. Die Verfahren, die bei Teamscale als Studienobjekt angewendet wurden, weichen von den Methoden für GitHub-Travis-Projekte an verschiedenen Stellen ab – so auch hinsichtlich der Auswahl der zu betrachtenden Builds.

Für die Entwicklung von Teamscale wird unter anderem GitLab eingesetzt, und dort werden auch die Tests ausgeführt. Builds werden in GitLab in einer „Pipeline" ausgeführt, in der zum Beispiel die Kompilierung des Codes, die Ausführung von Tests und das Bilden einer Distribution als „Jobs" ablaufen. In einem Betrachtungszeitraum von etwa 2000 Pipelines (75399-77695) wurden manuell 40 Builds herausgesucht, bei denen Tests fehlschlagen. Weit mehr Builds wurden gesichtet, da Builds, in denen der Test offensichtlich aufgrund von Infrastrukturfehlern fehlschlug, nicht in die nähere Betrachtung einbezogen wurden.

6.3 Mutationsanalysen

Mittels PITest[3] wurde pro Projekt eine vollständige Mutationsmatrix aufgestellt. Die Möglichkeit eine vollständige Matrix aufzustellen wurde jüngst von Niedermayr implementiert. In dieser Matrix ist für jeden Mutanten einer Methode festgehalten, welche Tests ihn aufdecken können und welche Testausführungen, die die entsprechende Methode überdecken, trotz Mutation nicht fehlschlagen.

Für insgesamt 14 GitHub-Travis-Projekte wurden im Benchmark Algorithmen, die Mutationsmatrizen als Eingabedaten verwenden, evaluiert.

Die Matrizen für 9 Projekte (bitcoinj, gson, jsoup, LittleProxy, biojava, graphhopper, traccar, commons-lang und geometry-api-java) wurden von Rainer Niedermayr bereitgestellt. Die Mutationsmatrizen für die 5 GitHub-Travis-Projekte dropwizard, graylog2-server, logback, retrofit und spring-data-examples wurden im Rahmen der vorliegenden Arbeit berechnet.

3 `http://pitest.org`

Dasselbe gilt auch für das Studienobjekt Teamscale. Da Teamscale *gradle* und nicht wie die GitHub-Travis-Projekte *maven* als Build-Werkzeug verwendet, wurde zur Erzeugung der Mutationsmatrix das gradle-pitest-plugin[4] angepasst. Die Anpassung war notwenig, da der derzeit veröffentlichte Stand nicht alle Möglichkeiten, die PITest in der aktuellen Version bereithält, in der Konfiguration zulässt. Zur erfolgreichen Mutationsanalyse waren auch Anpassungen in PITest selbst notwendig: Während der Testausführung löst das in Teamscale verwendete Bundle-System Pfade zu Klassenverzeichnissen auf. Das gradle-pitest-plugin startete die Tests jedoch mit kompilierten .jar-Dateien, so dass diese Auflösung fehlschlug. In PITest wurde das Problem behoben, indem der übergebene Buildpath von den .jar-Dateien auf die entsprechenden Klassenverzeichnisse geändert wurde.

Die im Rahmen dieser Arbeit erzeugten Mutationsmatrizen fanden auf einem Laptop (Dual Core i7, 2,9 Ghz, 8GB RAM) statt. Die Mutationsanalyse am Studienobjekt Teamscale dauerte etwas über 52 Stunden.

6.4 Statischer Aufrufgraph

Einer der untersuchten Priorisierungsalgorithmen nutzt statische Aufrufgraphen als Eingabe. Um diese zu erzeugen, wurde das Programm *java-callgraph*[5] verwendet. Es wurde so modifiziert, dass der Aufrufgraph in einer Datei abgelegt wird.

In der Priorisierung wird mithilfe des passenden Aufrufgraphen die Menge von Methoden berechnet, die durch einen Test aufgerufen werden.

6.5 Stack-Distanz von Testfall zu überdeckter Methode

Gegenwärtig untersucht Niedermayr den Zusammenhang von der Aufruf-Distanz (Stack-Distanz) von ausgeführtem Test zur überdeckten Methode und Testeffektivität (Preprint [NW] liegt vor). Für die hier vorliegende Studie waren bereits berechnete Daten von Niedermayr verfügbar gemacht worden. Diese wurden in unbearbeiteter Form verwendet.

6.6 Erhebung von Testausführungsinformationen

Um für die untersuchten Builds Coverage und weitere Testausführungsinformationen (Dauer, Testergbenis) aufzuzeichnen, wurde der Teamscale JaCoCo Agent verwendet. Damit die Aufzeichnung durch den Agent während der Testausführung stattfinden kann, musste die Maven-Projektkonfiguration jedes einzelnen untersuchten Builds angepasst werden. Dafür wurde ein Programm `PomAdjuster` in Java geschrieben, das im Projektverzeichnis nach Konfigurationsdateien (`pom.xml`) sucht. In diese werden im Konfigurationsabschnitt für die Testausführung (surefire und failsafe), der Agent in die Java-Argline eingefügt und die Testlistener für die Testframeworks JUnit4 und TestNG eingerichtet. Die Konfiguration des TestListeners für JUnit5 Tests erfolgte davon abweichend nicht in der `pom.xml`, sondern nach der Übersetzung des Programms durch das Hinzufügen einer entsprechenden Datei im Ordner `/META-INF/services/`. Die Entwicklung des `PomAdjuster`s ermöglichte die automatische Projektkonfiguration für die Coverageerhebung und war für die Machbarkeit der Studie von großer Bedeutung.

4 `https://gradle-pitest-plugin.solidsoft.info`
5 `https://github.com/gousiosg/java-callgraph`

Die Erhebung der Coverage für die untersuchten Builds selbst ist mit einem hohem Rechen- und Zeitaufwand verbunden. Um trotzdem in einem angemessenen Zeitrahmen Daten erzeugen zu können, wurde der Build- und Testprozess auf virtuellen Serverinstanzen von Google (Compute Engine) ausgelagert.

Auf fünf verschiedenen Instanzen, die mit je 26 Prozessorkernen und 105GB Arbeitsspeicher eingerichtet waren, wurden parallel je 17 Builds gestartet. Um auch diesen Prozess zu automatisieren, wurde ein Programm implementiert, das ein Bash-Skript zur Einrichtung und Ausführung der Builds startet. Durch die Verwendung von virtuellen Konsolenfenstern (GNU Screen[6]) lief dieser Prozess im Hintergrund ab, konnte bei Bedarf aber auch manuell überwacht werden. Zusätzlich wurden Statusinformationen automatisiert in einer Datenbank abgelegt.

Nach Abschluss der Aufzeichnung wurden die Ordner, die den Quelltext, die kompilierten Klassen und die Testausführungsinformationen enthielten, in ein Archiv gepackt, um es für die weitere Verwendung bereitzuhalten.

Die Coverageerhebung für das System unter Test „Teamscale“ fand nicht auf einer Google Instanz statt, sondern auf einem Laptop.

6 `http://www.gnu.org/software/screen/screen.html`

7 Aufbau der Studie

Um diese Studie nachvollziehbar zu machen, wird im Folgenden gezeigt, welche Schritte zu welchem Zeitpunkt an welchen Angriffsflächen getätigt wurden.

7.1 Studienobjekte

Im Rahmen der Studie wurde die Leistungsfähigkeit von verschiedenen *Priorisierungsverfahren* hinsichtlich der Zeit bis zum ersten Testfehlschlag anhand von *Builds* unterschiedlicher *Projekte* untersucht.

7.1.1 Untersuchte Priorisierungsverfahren

Die Menge der zu untersuchenden Priorisierungsverfahren ergab sich aus Forschungen an der Forschungsgruppe, aktuellen Forschungsaktivitäten von Rainer Niedermayr und Verfahren, die auf einfacheren Metriken basieren.

Die acht untersuchten Priorisierungsverfahren (zuzüglich Variationen) werden im Laufe dieser Studie anhand von Kürzeln identifiziert. Sie werden in der folgenden Aufzählung vorgestellt.

ACPTPS steht kurz für die **A**dditional **C**overage **P**er **T**ime **P**rioritization **S**trategy. Dieses Verfahren sortiert rundenweise die Tests nach ihrer zusätzlichen Abdeckungsgeschwindigkeit, die sich aus den Testdaten der letztbekannten Ausführung errechnet. „Zusätzlich“ bezieht sich hier auf die bisher gewählten Tests – deckt also ein Test nur wenige Teile des Systems ab, die vorher nicht von einem bereits selektierten Test in der selben Runde abgedeckt wurden, so ist die zusätzliche Abdeckungsgeschwindigkeit niedrig. Tests, die einen geänderten Programmteil abdecken, werden in der Ausführungsliste weiter nach vorne priorisiert. Eine Runde endet, wenn kein noch nicht priorisierter Testfall zusätzliche Teile des Systems abdeckt.

Dieses Verfahren entspricht dem aktuell in Teamscale implementierten Standardverfahren zur Testselektion und -priorisierung unter Setzen der Option `includeNonImpacted=true`, wodurch keine Selektion stattfindet.

EACPTS **E**nhanced **A**dditional **C**overage **P**er **T**ime **S**trategy. Dieses Verfahren nutzt das aktuell in Teamscale implementierte Standardverfahren zur Selektion und Priorisierung von Testfällen auf Basis von TIA. Die Test-Selektion bewirkt, dass nur Tests priorisiert werden, die Änderungen im Code-Delta abdecken oder neu sind ($\hat{=}$ relevante Tests).

Die EACPTS ist ebenfalls ein rundenbasiertes Verfahren. In jeder Runde werden so lange Tests ausgewählt, bis alle Änderungen im Code-Delta abgedeckt sind. Dabei werden vor jeder Testauswahl aus den relevanten (siehe oben) und noch nicht priorisierten Tests, die Tests nach ihrer zusätzlichen Abdeckungsgeschwindigkeit sortiert. Der relevante Test mit der höchsten zusätzlichen Abdeckungsgeschwindigkeit wird dann der Ausführungsliste hinzugefügt.

SDPS **S**tring **D**istance **P**rioritization **S**trategy. Anhand der Quelltexte der Testmethoden wird eine Distanzmatrix aufgebaut. Zuerst wird derjenige Testfall ausgeführt, der

die höchste Stringdistanz zu allen anderen Tests hat. Anschließend wird immer der Testfall als nächstes ausgeführt, der zum eben ausgeführten Testfall die höchste Stringdistanz aufweist.

RBIFPS **R**andom **B**ut **I**mpacted **T**ests **F**irst **S**trategy. Dieses Verfahren war bereits vor Studienbeginn in Teamscale implementiert. Es wird sowohl eine Menge von Tests gebildet, die entweder eine Änderung im Code-Delta abdecken oder neu sind (Menge 1), als auch die Menge der restlichen Tests im System (Menge 2). Beide Mengen werden jeweils (nicht zusammen) gemischt und anschließend nacheinander zur Ausführung vorgeschlagen – Menge 1 zuerst, dann Menge 2.

DPS **D**uration **P**rioritization **S**trategy. Von dieser Strategie werden die Tests nach ihrer letztbekannten Ausführungsdauer sortiert. Schnelle Tests werden in der Ausführungsliste nach vorne platziert.

FRPS **F**ully **R**andom **P**rioritization **S**trategy. Die Tests werden in einer wahllosen Reihenfolge zur Ausführung vorgeschlagen.

MPTPS **M**ethods **P**er **T**ime. Diese Strategie ordnet die Tests nach ihrer Abdeckungsgeschwindigkeit (ungeachtet bereits ausgeführter Methoden). Tests, die viele Methoden in kurzer Zeit abdecken, werden zuerst ausgeführt.

TNCPS **T**otal **N**umber **C**allees **P**rioritization **S**trategy. Anhand eines statischen Aufrufgraphen wird für jeden Testfall berechnet, wie viele Methodenaufrufe durch ihn erfolgen. Testfälle, die demnach viele Methoden aufrufen, werden nach vorne priorisiert.

Zusatz „t“ Der Folgebuchstabe **t** steht für die zusätzliche Bewertung von überdeckten Methoden hinsichtlich ihrer Trivialität. Die Prämisse hier ist, dass Fehler in kurzen und einfachen Methoden weniger wahrscheinlich sind und eine solche Methode aus diesem Grund weniger testenswert ist. Eine überdeckte Methode, die als trivial eingestuft wurde, wird in der Priorisierung als weniger wichtig auszuführen betrachtet.

Zusatz „m“ Folgt dem Kürzel EACPTS der Zusatz **m**, so flossen Daten aus Mutationsanlysen in die Testpriorisierung ein. Die verwendeten Mutationsmatrizen enthält für jede mutierte Methode die Menge an Tests, die die Mutation aufdecken. Führt ein Test eine Methode aus, schlägt aber trotz einer eingestreuten Mutation nicht fehl, so testet der Test die Methode nur scheinbar [NJW16]. Siehe auch Abschnitt 6.3.

Zusatz „s“ Tritt die EACPTS gefolgt von dem Buchstaben **s** auf, so bedeutet das, dass die Daten einer Stack-Distanz Analyse die Priorisierung beeinflusst haben. Hier wird davon ausgegangen, dass Methoden, die beim Ausführen des Testfalls im Aufruf-Stack weiter von der Testfallmethode entfernt sind, weniger effektiv getestet werden. Durch viele Methoden-Aufrufe dazwischen werden Methoden mit einer höheren Stack-Distanz nur *indirekt* getestet und die Wahrscheinlichkeit, einen Fehler in der entfernten Methode aufzudecken wird niedriger eingeschätzt. Eine mittels Machine Learning gelernte Wahrscheinlichkeit, dass eine Methode-Testfall-Beziehung aufgrund ihrer Stack-Distanz und weiteren Daten scheingetestet ist, wird in der Prioriserung entsprechend berücksichtigt. [NRW18a; NRW18b]

Suffix nach „–“ Der Suffix nach dem Zeichen „–“ beschreibt die bei der Priorisierung eingesetzte Realitätskonfiguration. Die möglichen Konfigurationen sind in Abschnitt 8.3.1 aufgelistet und beschrieben.

Die Datenlage erlaubt es – insbesondere für Algorithmen, die Ergebnisse zusätzlicher Analysen als Eingabedaten benötigen – nicht, jeden Algorithmus in jedem Projekt und auf jeden Build anzuwenden.

7.1.2 Untersuchte Projekte

Die Projektauswahl erfolgte anhand des Abgleichs von Studienobjekten aus der Vorgängerstudie und der Menge an Projekten, für die durch Niedermayr Mutationsdaten und Stack-Distanz-Daten vorlagen. Eine Übersicht über die beteiligten Studienprojekte und die Teilbereiche des Benchmarks, in denen sie untersucht wurden, ist in Tabelle 7.1 gegeben.

7.1.3 Untersuchte Builds

Hinsichtlich der Fragestellungen RQ 1.x wurden sowohl Builds, die bereits in der Vorgängerstudie untersucht wurden, als auch Builds von denselben Projekten, die zu einem späteren Zeitpunkt stattfanden, ausgewählt. Bei den Builds aus der Vorgängerstudie wurde sich auf diejenigen beschränkt, in denen die Testfehlschläge vermutlich auf Codeänderungen beruhen und der Algorithmus der Forschungsgruppe den fehlschlagenden Test zur Ausführung vorschlug (vgl. Abschnitt 4.2). Mehr zum Prozess der Buildauswahl in Abschnitt 6.1.

Eine Beschreibung, welche acht Builds des Studienobjekts *Teamscale* zum Einsatz kamen, wird in Abschnitt 6.2 gegeben.

Für die Forschungsfragen RQ 2.x wurden die Codestände verwendet, auf deren Basis die Mutationsanalysen von Niedermayr durchgeführt wurden und zu denen daher die Mutationsmatrizen passen. Für Projekte, für die im Rahmen der hier vorliegenden Studie selbst Mutationsmatrizen (vgl. 6.3) erzeugt wurden, wurde ein beliebiger grüner Build aus etwa Mitte 2018 gewählt.

7.2 Design der Studie

In diesem Abschnitt wird vorgestellt, wie man von den aus der Studie gewonnenen Daten zu einer Beantwortung der Forschungsfragen gelangt.

Benchmark mit real aufgetretenen Testfehlschlägen (RQ 1.x)

RQ 1.1: Welcher Algorithmus führt durchschnittlich zur kürzesten Time to Failure? Durch Analyse der Mediane und oberer Quartile wird festgestellt, welcher Algorithmus am besten hinsichtlich der Time to Failure arbeitet.

RQ 1.2: Wie hoch ist der Anteil an impacted Tests vom letzten erfolgreichen bis zum fehlschlagenden Build? Die quantitative Fragestellung wird durch die Darstellung der erhobenen Daten beantwortet.

RQ 1.3: Häufigkeit von unentdeckten Testfehlschlägen durch die Selektierung von Testfällen mit dem vorgeschlagenen Verfahren und optionalen Anpassungen. Die quantitative Fragestellung wird durch die Darstellung der erhobenen Daten beantwortet.

Mutations-Benchmark (RQ 2.x) Die Fragestellungen RQ 1.x und RQ 2.x sind äquivalent und werden lediglich auf einer anderen Datenbasis berechnet. Die Beantwortung der Forschungsfragen ergibt sich also auf denselben Wegen wie für die Fragen RQ 1.x.

Das Changeset, das beim Mutations-Benchmark neben der mutierten Methode zusätzliche Methoden enthalten soll, wird durch eine Implementation von Niedermayr berechnet. Sie wählt die zusätzlichen Methoden nicht wahllos aus, sondern fügt Methoden hinzu, die sich im „näheren Umfeld“ (in dem eine zusätzliche Veränderung auch bei einem realen Commit plausibel ist) zum Mutanten befinden.

Tabelle 7.1: Studienobjekte und ihre Teilnahme an der Beantwortung einzelner Forschungsfragen. Projekte, die sowohl am Benchmark mit real fehlgeschlagenen Tests als auch am Mutation-Based-Benchmark beteiligt waren, sind zuoberst aufgelistet, gefolgt von Projekten, die nur für einen der beiden Benchmarks als Datengrundlage dienten. Für den Fragenkomplex RQ 1.x konnten Daten von insgesamt 201 verschiedenen Builds verwendet werden.

Studienobjekt	Multi-Modul-Projekt	Studienobjekt für RQ 1.x	Untersuchte Builds in RQ 1.x	Studienobjekt für RQ 2.x	Anzahl kLOC (ca.)	Anzahl src-kLOC (ca.)	Anzahl test-kLOC (ca.)	∅ Anzahl Tests
adamfisk / LittleProxy		✓	4	✓	14,4	8,0	6,4	184
apache / commons-lang		✓	7	✓	142,9	77,1	65,8	4060
biojava / biojava	✓	✓	42	✓	290,4	244,9	45,5	1133
cqse / teamscale	✓	✓	8	✓	869,9	763,8	106,1	6410
dropwizard / dropwizard	✓	✓	2	✓	58,6	28,0	30,6	1202
graphhopper / graphhopper	✓	✓	30	✓	72,7	48,0	24,7	1463
google / gson		✓	2	✓	38,1	18,0	20,1	1039
square / retrofit	✓	✓	12	✓	24,6	9,9	14,8	540
traccar / traccar		✓	3	✓	80,7	67,4	13,4	302
apache / pdfbox	✓	✓	2		232,4	207,6	24,8	1192
apache / jackrabbit-oak	✓	✓	2		431,9	255,6	176,3	6328
aws / aws-sdk-java	✓	✓	20		4496,6	4459,5	37,1	1027
doanduyhai / Achilles	✓	✓	12		72,4	41,6	30,8	281
Graylog2 / graylog2-server	✓	✓	23		140,2	105,7	34,4	1208
jamesagnew / hapi-fhir	✓	✓	4		2049,3	1847,4	201,9	2783
jmxtrans / jmxtrans	✓	✓	1		25,8	16,6	9,2	234
openpnp / openpnp		✓	6		74,5	73,3	1,2	11
OpenTSDB / opentsdb	✓	✓	2		152,1	75,2	76,9	387
OfficeDev / ews-java-api		✓	1		118,6	115,4	3,1	114
rackerlabs / blueflood	✓	✓	4		64,2	31,2	33,0	755
spullara / mustache.java	✓	✓	3		10,4	5,5	5,0	191
timmolter / XChange	✓	✓	2		157,4	135,1	22,3	5
twilio / twilio		✓	2		104,9	87,0	17,8	769
undera / jmeter-plugins	✓	✓	7		73,6	40,2	33,4	1063
bitcoinj / bitcoinj				✓	156,4	131,2	25,2	5296
Esri / geometry-java-api				✓	112,6	87,3	25,3	407
jhy / jsoup				✓	28,1	18,2	9,9	658
qos-ch / logback	✓			✓	99,8	54,5	45,3	1013

Weitere Fragestellungen (RQ 3.x)

RQ 3.1: Wie wirken sich Beschränkungen in der Ausführungsreihenfolge von Tests (z. B. innerhalb von Modulen oder Klassen) auf die Time to Failure aus? Median und obere Quartile der Performanz des Algorithmus EACPTS werden über die verschiedenen Realitätsoptionen hinweg verglichen. Es sollen die Daten von GitHub-Travis-Projekten aus dem Benchmark mit real fehlgeschlagenen Tests verwendet werden.

RQ 3.2: Wie häufig und langandauernd sind klasseneigene Setup- und Teardown-Methoden? Die aufgezeichneten @BeforeClass- und @AfterClass-Methoden werden für jedes Projekt in absoluten Zahlen der Anzahl aller Tests und der Gesamttestdauer gegenübergestellt. Es sollen die Daten von GitHub-Travis-Projekten aus dem Benchmark mit real fehlgeschlagenen Tests verwendet werden.

RQ 3.3: Was sind die Auswirkungen einer unregelmäßigen Aktualisierung von Test-zu–Methoden-Beziehungen? Es werden die ursprünglich gemessenen Time to Failure Zeiten den unter Verwendung von älteren Testausführungsinformationen neu berechneten Zeiten gegenübergestellt.

7.3 Methodik

In diesem Abschnitt wird zunächst beschrieben, wie das Setup allgemein vorbereitet wurde. In eigenen Absätzen zu jeder Forschungsfrage wird im Anschluss daran das weitere Vorgehen beschrieben.

1. Durch die Analyse von Travis CI Buildlog Daten wurden Builds identifiziert, die aufgrund von Testfehlern auf der Infrastruktur von Travis CI fehlschlugen („roter Build“, siehe auch Abschnitt 6.1).
2. Der Commit, der am nächsten zum Codestand des fehlschlagenden Builds liegt, für den allerdings der Build erfolgreich war, wurde durch die Analyse von Repositorydaten von GitHub ausgemacht („grüner Build“, siehe ebenfalls Abschnitt 6.1).
3. Beide Builds wurden heruntergeladen und Testausführungsdaten (Coverage, Dauer und Testresultate) erhoben (Abschnitt 6.6). Die Ausführungsdaten des grünen Builds dienten der eigentlichen Test-Impact-Analyse und die des roten Builds zur Bildung der Mengen von hinzugefügten und geänderten Tests (zu neuen und geänderten Tests mehr in Abschnitt 4.3.5).
4. Für den zu untersuchenden Build wurde ein Teamscale Projekt angelegt, das den Codestand zum roten und grünen Build und die Testausführungsdaten zum grünen Build enthielt.
5. Der Teamscale-Service zur Berechnung einer geordneten Testausführungsliste, die auf Basis der im letzten Schritt erwähnten Daten stattfand, wurde für jeden zu evaluierenden Priorisierungalgorithmus aufgerufen. Eine Übersicht ist in Abb. 7.1 gegeben.
6. Für jede empfangene Testausführungsliste wurde die Zeit berechnet, die es braucht, bis die Ausführung einen bekanntermaßen fehlschlagenden Test erreicht. Dies ist die Time to Failure.

7.3.1 Was sind die Auswirkungen einer unregelmäßigen Aktualisierung von Test-zu-Methoden-Beziehungen?

Für das Projekt *graphhopper* wurde die Berechnung der Times to Failure wiederholt. Die Historienaufbau in Teamscale wurde allerdings derart verändert, dass nun vor dem

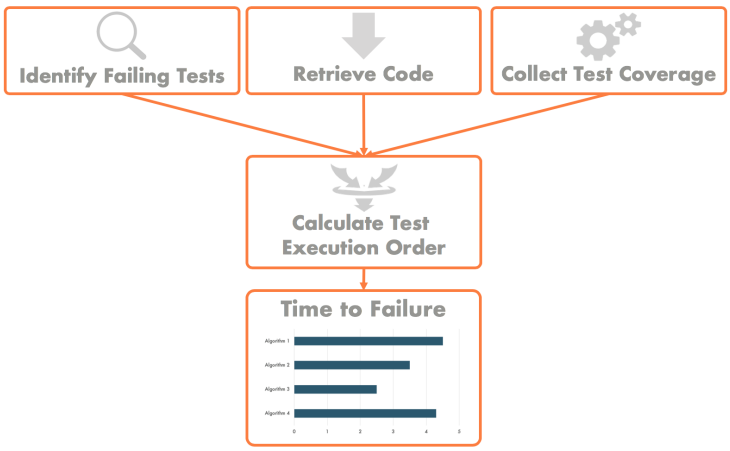


Abbildung 7.1: Diese grobe Übersicht zeigt die für die Berechnung der Time to Failure verwendeten Daten und Verarbeitungsschritte.

Tabelle 7.2: Es sind die Paare von fehlgeschlagenen Commits und die dazu in der Forschungsfrage RQ 1.1 verwendeten Parent-Commits aufgelistet, sowie das dazwischenliegende Intervall. Die letzte Tabellenspalte gibt das Intervall vom ältesten in der Studie verwendeten Parentcommit '2a38d6' zum fehlgeschlagenen Commit an.

Roter Build	Parent	Parent→Rot	Ältester→Rot
041eb9	e2482e	0h	645h (26d)
0aa491	1895cf	0h	3018h (125d)
0c7a17	a54b6f	1h	3049h (127d)
6bb467	e8fff1	3h	6022h (250d)
944723	e8fff1	4h	6023h (250d)
b83a85	e8fff1	4h	6023h (250d)
ab9a21	f47162	0h	6211h (258d)
c59ffc	68610e	3h	6211h (258d)
282a3f	68610e	3h	6210h (258d)
7cb7da	35cea1	27h	7897h (329d)
c6677d	9c844a	48h	8951h (372d)
457e07	1717cd	123h	12791h (532d)
205255	c8b4a5	97h	12934h (538d)
850208	1717cd	261h	12929h (538d)
f70bc5	c8b4a5	92h	12929h (538d)

grünen Parent-Commit (dessen Testausführungsinformationen in der Forschungsfrage RQ 1.1 für die TIA verwendet wurde) der Codestand des ältesten im Benchmark verwendeten Parent-Commit liegt. Ebenfalls wurden die zum ältesten Codestand gehörigen Testausführungsinformationen in Teamscale abgelegt, jedoch nicht mehr diejenigen für den grünen Build.

Damit wird das „alte" Mapping von Tests zu überdecken Methoden genutzt, aber weiterhin das Code-Delta zwischen grünem und rotem Build in der Priorisierung verwendet.

In Tabelle 7.2 ist aufgelistet, wie viel Zeit zwischen grünem und rotem Build und wie viel Zeit zwischen dem ältesten Build und dem roten Build liegt.

7.3.2 Zusätzliche Eingabedaten für Priorisierungsstrategien

Abgesehen vom Code-Delta und Informationen zu einer vergangenen Codeausführung, kann die Logik eines Priorisierungsalgorithmus durch das Hinzufügen weiterer Eingabedaten erweitert werden. Im Fall der hier vorliegenden Studie wurden vier weitere Aspekte berücksichtigt.

- Klassifikation trivialer Methoden: Eine Implementation in Teamscale ist durch die

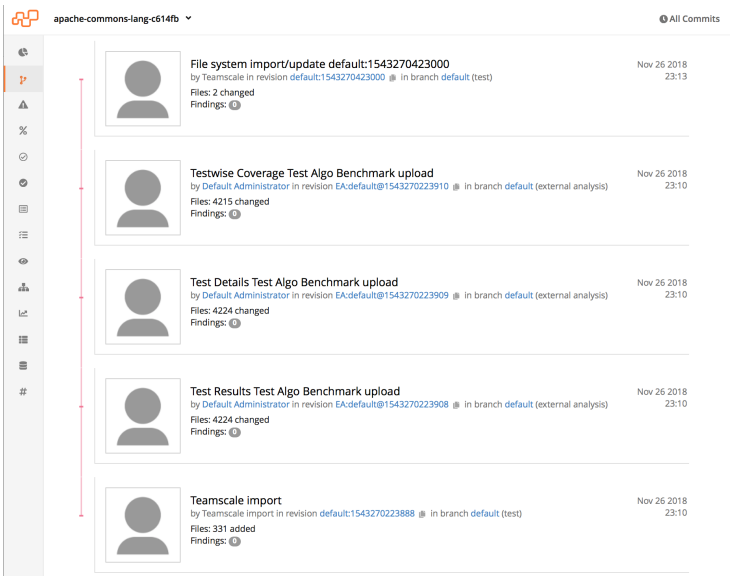


Abbildung 7.2: Die Aktivitätsanzeige in Teamscale zeigt (in der Abbildung ganz unten) den grünen Commit als „Teamscale import“ an, darauf folgen die Uploads für die Testausführungsdaten und zuletzt die Aktualisierung auf den zweiten Codestand. Die sichtbaren Zeiten entsprechen nicht den tatsächlichen Commitzeiten, sondern wurden für die Bearbeitung dieser Arbeit frei gewählt und vom automatisierten Ablauf für jedes Projekt verwendet.

Forschung von Niedermayr [NRW18a; NRW18b] vorhanden.

- Scheingeteste Methoden: Die Herkunft der Daten aus Mutationsdaten für die Einordnung scheingetesteter Methoden ist in Abschnitt 6.3 erläutert.
- Stack-Distanz: Wie in Abschnitt 6.5 beschrieben, konnten diese Daten aus der Forschung von Niedermayr übernommen werden.
- Der statische Aufrufgraph wurde im Zuge des Benchmarks berechnet, vgl. Abschnitt 6.4.

7.3.3 Vorbereitung eines Teamscale Projekts pro zu untersuchenden Build

Dieses Programm ruft aus einer Datenbank ab, welche fehlschlagenden Builds es für welche Projekte gibt (vgl. weiter oben, Abschnitt 6.1). Für jeden dieser Builds wird überprüft, ob die für die Analyse notwendigen Archive (Code, kompilierte Klassen, Testausführungsdaten; vgl. Abschnitt 6.6) existieren. Wenn dies zutrifft, wird auf einer laufenden Teamscale-Instanz ein dazugehöriges Projekt erstellt. Diesem werden die beiden Code-Revisionen sowie die Testausführungsdaten für die erste Coderevision (für die es – zur Erinnerung – einen grünen Build auf Travis CI gab) zugeführt. Um eine gute Übersichtlichkeit in den Projekten zu gewähren, wurde nicht das Git-Repository mit Teamscale verknüpft, sondern Multi-Version-Filesystem-Konnektoren verwendet. Diese lesen den Quelltext der beiden Versionen (Codestand zum grünen und roten Build) aus vorbereiteten Ordnern aus. Die Ordner tragen als Namen den Zeitstempel, der für den Importzeitpunkt in Teamscale verwendet werden soll. Die Teamscale-Aktivitätsansicht für ein so vorbereitetes Projekt ist in Bild 7.2 gegeben.

Es wurde im Vorhinein ein Analyseprofil für die Sprache Java angelegt, in der für die Studie unnötige Optionen abgeschaltet wurden, um die Rechenzeit gering zu halten.

Ähnlich wie bei der Erhebung der Testausführungsinformationen fand die Berechnung der Times to Failure auf einer Google-Instanz statt. Für die gleichzeitige Analyse vieler

zu untersuchender Builds wurde die Konfiguration auf 54 Prozessorkerne und 600GB Arbeitsspeicher festgelegt.

Die eingesetzte Teamscaleversion enthielt die in Kapitel 4 beschriebenen TIA-Funktionalitäten und Umbauten/Erweiterungen für diese Studie. Um Teamscale auf der Google-Instanz einsetzen zu können, wurde vom Entwicklungsstrang des Git-Repositorys eine Distribution erzeugt, die einen Docker-Container verwendet. Diese wurde zwischenzeitlich auf der Plattform Docker Hub abgelegt. Die nun dort verfügbare Distribution wurde von der Google-Instanz aus heruntergeladen und im Anschluss installiert.

Der Zeitpunkt nach Abschluss der Projekteinrichtung in Teamscale wurde auch dazu genutzt, um jene Builds von der Analyse auszuschließen, die keine Codeänderung in Java vom grünen bis zum roten Build erfahren haben. Die Konstellation in der Teamscale-Aktivitätsübersicht und ein Beispiel für einen codefreien Commit sind im Anhang auf Seite 63 gegeben.

7.3.4 Abruf der geordneten Testausführungslisten und Berechnung der Time to Failure

Im Grundlagen-Kapitel und in Kapitel 4 wurden das Tool Teamscale und die darin implementierte Test Impact Analyse sowie die Nutzung der Daten für eine Testpriorisierung und -selektion vorgestellt.

Für die Erhebung der Time to Failure wurde ein Java-Programm `TestPrioritization-Benchmark` entwickelt. Dieses bereitet zum einen ein Teamscale Projekt vor und ruft danach für die zu untersuchenden Builds und Priorisierungsalgorithmen die sortierte Testausführungsliste ab. Auf Basis letzterer erfolgt im Programm die Berechnung der Time to Failure. Nach der erfolgreichen Analyse in Teamscale wird für jeden zu untersuchenden Build und die zu untersuchenden Priorisierungsalgorithmen eine HTTP-Anfrage an den Service von Teamscale gestellt, der priorisierte Testlisten zurückgibt. Die Time to Failure ergibt sich aus der Summe der Ausführungszeiten der Tests, die vor einem bekanntermaßen (zum Beispiel in einem Travis Build) fehlgeschlagenen Tests ausgeführt werden sollen, und der Ausführungszeit des fehlschlagenden Tests.

Sind mehrere scheiternde Tests bekannt, so ergibt sich bei unterschiedlichen Priorisierungsalgoritmen die Time to Failure möglicherweise durch das Erreichen verschiedener Tests. Schlugen zum Beispiel in einem Build in der Vergangenheit die beiden Testfälle `ATest` und `ZTest` fehl, so können zwei (wohlgemerkt fiktive) Priorisierungsalgorithmen, die Tests auf- oder absteigend alphabetisch sortieren, beide gute Time to Failure-Werte erreichen, obgleich sie unterschiedliche Tests aufdecken.

In der entworfenen Datenstruktur werden die Times to Failure für jeden auf Travis CI fehlschlagenden Test gespeichert. Sowohl die priorisierten Testlisten je Algorithmus als auch die Time to Failure-Werte wurden in einer MySQL-Datenbank abgelegt.

Listing 7.1: Pseudocode zur Bestimmung der Time to Failure.

```
param project: Project
param prioritizationStrategyToUse: IPrioritizationStrategy
param knownFailingTests: Set<TestIdentifier>

List<TestDetailsForPrioritization> orderedTests =
  retrieveOrderedTestList(project.name, prioritizationStrategyToUse);

int timeToFailureInMs = 0;

foreach executedTest in orderedTests {
    timeToFailureInMs += executedTest.getDurationInMs();

    if (knownFailingTests.containsTestWithIdentifier(executedTest)) {
        return timeToFailureInMs;
    }
}

return TEST_NOT_REACHED;
```

8 Implementierung

Dieses Kapitel beschreibt Implementierungen, die im Rahmen der Studie getätigt wurden. Dies waren einerseits Anpassungen am bestehenden TIA System (siehe dazu auch Kapitel 4), andererseits Werkzeuge und Ansichten, die zur Bewältigung spezieller Teilaufgaben in der Studie dienten.

8.1 Unterstützung von @BeforeClass- und @AfterClass-Methoden

In früheren Versionen des Werkzeugs zur Aufnahme von Testausführungsinformationen, wurde die Aufnahme exakt zu Beginn eines einzelnen Testfall gestartet und endete zeitgleich mit dem Testfall. Aus diesem Grund standen zum Beispiel Coveragedaten aus klassenspezifischen Setup- und Teardown-Methoden nicht in der TIA zur Verfügung.

Zur Unterstützung der Aufzeichnung von Ausführungsdauer und Coverage von @BeforeClass- und @AfterClass-Methoden wurden der Teamscale Jacoco Agent und der Test-Listener angepasst. Wird eine JUnit-Testsuite über das maven-surefire-Plugin ausgeführt, so kann die Aufzeichnung zwischen dem letzten Testfall einer Testklasse und dem ersten Testfall einer nächsten Klasse oft nicht eindeutig zugeordnet werden. Nach Abschluss des letzten Testfalls der einen Klasse wird im Listener die Methode `testFinished` aufgerufen. Vor Beginn des ersten Testfalls in der nächsten Testklasse wird die Methode `testStarted` aufgerufen. Da zwischen den beiden Aufrufen des Listeners sowohl die @AfterClass-Methode der ersten Klasse als auch die @BeforeClass-Methode der folgenden Klasse ausgeführt werden kann und der Agent dies ununterbrochen aufnimmt, kann nicht zugeordnet werden, zu welcher Klasse die aufgenommene Information zu rechnen ist. Im Rahmen dieser Arbeit trägt dieser Teil der Ausführungsinformation den Namen „implizit zugeordnete/aufgenommene Before- bzw. After-Coverage“.

Um die Information trotzdem zu verarbeiten, wurde der Agent in der Art verändert, dass die bisher nicht verfügbare Ausführungsreihenfolge der Tests nachvollziehbar ist (die Datenstruktur wurde um ein Feld für den Vorgängertest erweitert). Nicht eindeutig zuordenbare Coverage wird nun sowohl mit der beschriebenen ersten als auch der nachfolgenden Testklasse verknüpft.

Insgesamt ist die beschriebene hinzugefügte Unterstützung eine Verbesserung der TIA, denn Änderungen an Methoden, die ausschließlich von @BeforeClass- oder @AfterClass-Methoden ausgeführt wurden, führten bisher nicht zum Vorschlag eines entsprechenden Tests. In der neuen Version werden alle Testfälle der betreffenden Klasse ausgeführt.

Während dieser Untersuchung wurde mit positivem Ergebnis ebenfalls überprüft, ob bei der Ausführungsaufzeichnung @Before- und @After-Methoden, die unmittelbar vor einem Testfall ausgeführt werden, vom Javaagent korrekt aufgezeichnet werden.

8.2 Konverter für testspezifische Coverage

Sowohl die Berechnung von als auch der Umgang mit testspezifischer Coverage fanden für die hier vorliegende Arbeit mit Mitteln aus Hand der CQSE GmbH statt. Das System

befand sich im Studienzeitraum in Entwicklung und unterlag mehrmaligen Formatänderungen.

Die Aufzeichnung von testspezifischer Coverage für Teamscale (als System unter Test) mit einem von der CQSE jüngst entwickelten Setup liefert Daten im neuesten Format. Der im Benchmark verwendete Versionsstand zur Analyse der Testdaten arbeitet hingegen mit einem älteren Format. Um die für Teamscale berechnete testspezifische Coverage trotzdem verwenden zu können, wurde ein Konverter entwickelt, der die Daten entsprechend übersetzt und die Kompatibilität herstellt.

8.3 Darstellung der erhobenen Daten

In der Vorgängerarbeit [Rot18] wurden aus den in einer Datenbank gespeicherten Times to Failure automatisch LaTeX-Dokumente erzeugt, die die Daten in Boxplots darstellten. Dieses System war verhältnismäßig unflexibel und wurde durch eine Neuentwicklung ersetzt.

In Teamscale wurde eine Ansicht implementiert, die Benchmarkergebnisse sowohl über alle untersuchten Builds (vgl. Abb. 8.1), über Builds eines bestimmten Projekts und für einen einzelnen Build (vgl. Abb. 8.2) anzeigen kann.

Die Gesamtübersicht zeigt die Time to Failure-Werte pro Algorithmus in Boxplots[1]. Darunter ist ein Balkendiagramm abgebildet, das die Zeitersparnis pro untersuchten Commit angibt, wenn die EACPTS anstatt einer zufälligen Testausführungsreihenfolge verwendet wird. Außerdem verweist die Seite auf die einzelnen Projektübersichten. Die Projektansicht ist der Gesamtübersicht im Aufbau sehr ähnlich, weshalb auf eine Abbildung verzichtet wird.

In der Ansicht für einen einzelnen untersuchen Build werden zuoberst Informationen zum Build (Datum, Status der für diesen Build ausgeführten Travis Jobs, fehlschlagende Tests im Travis Build) angezeigt. Diese Informationen werden teils von Travis abgerufen, teils stammen sie aus der studieneigenen Datenbank. Auf der Seite werden außerdem die Time to Failures für alle abgerufenen Algorithmen aufgelistet und dabei auch die verwendeten Parameter angezeigt. Im unteren Bereich der Seite erfolgt eine Anzeige, für wie viele Tests je im grünen und roten Build bei der Erhebung der Testausführungsinformationen eine Aufzeichnung erfolgte und wie viele Tests davon fehlschlugen. Zudem wird angezeigt, welche Tests (im roten Build) sowohl bei Travis als auch im Rahmen der Datenerhebung für die Studie fehlschlugen und wie viele Tests vom grünen bis zum roten Build hinzugefügt oder gelöscht wurden.

Der Speicherung der Benchmarkresultate liegt eine MySQL-Datenbank zugrunde, aus der die Daten für die Anzeigeseiten auf Teamscale abgerufen werden. Allerdings wurde die Datenstruktur im Vergleich zur Vorgängerarbeit angepasst.

8.3.1 Realitätsannäherung bei der Berechnung der Testausführungsdauer

In der Vorgängerarbeit [Rot18] wurde ein Prototyp für ein Benchmarksystem für Testpriorisierungsverfahren entwickelt. Die durchgeführte Priorisierung erfolgte ohne Beachtung mehrerer möglicherweise für die Time to Failure-Bestimmung relevanter Eigenschaften des testausführenden Systems:

1. Bei Multi-Modul-Projekten wurde nicht berücksichtigt, aus welchem Modul ein zu priorisierender Test stammt.

1 Zur Erzeugung der Boxplots wurde plotly.js verwendet `https://plot.ly/javascript/`

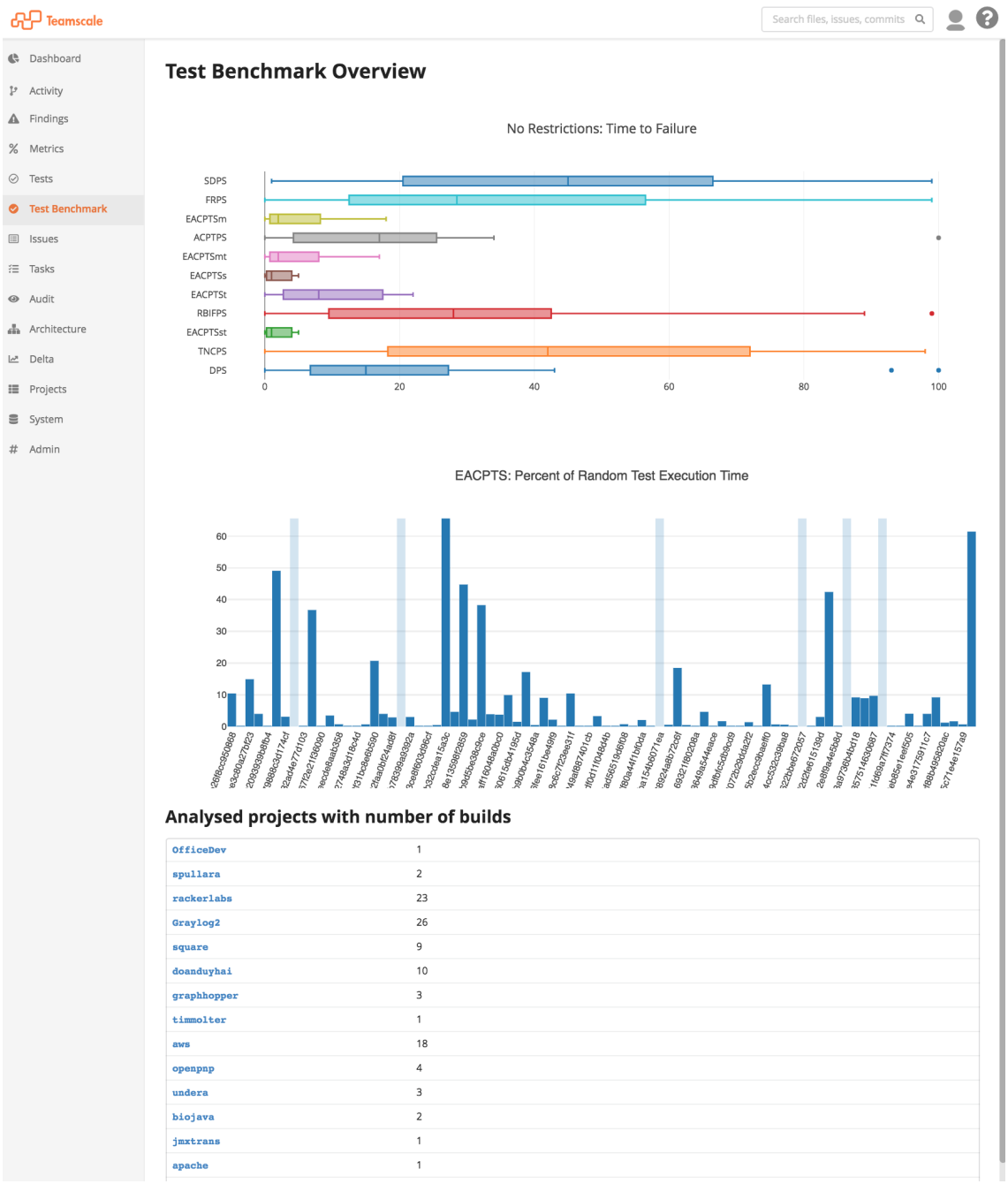


Abbildung 8.1: In Teamscale wurde eine Ansicht implementiert, die eine Übersicht über die Performanz der unterschiedlichen Priorisierungsalgorithmen bereithält. Es werden Boxplots, die die Time to Failure beschreiben dargestellt und in einem Balkendiagramm veranschaulicht, wie viel Testzeit (bis zum Auftreten eines Testfehlers) sich durch die Verwendung von EACPTS im Vergleich zu einer zufälligen Testreihenfolge sparen lässt. Die Auflistung der unterschiedlichen Projekte ermöglicht die Navigation zur projektspezifischen Übersichtsseite, die bis auf die Datenfilterung mit der allgemeinen Übersicht übereinstimmt und deshalb nicht im Einzelnen abgebildet ist.

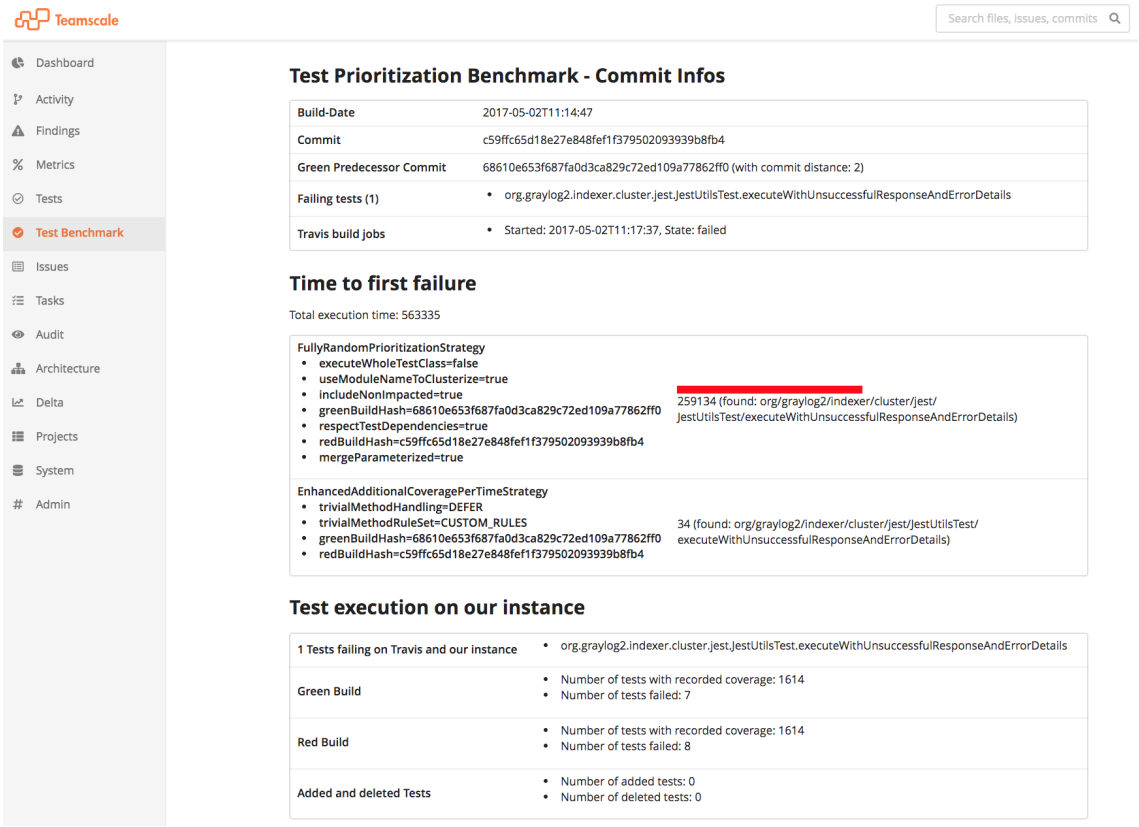


Abbildung 8.2: Neben der Gesamtübersicht und einer projektspezifischen Ansicht wurde auch eine Übersicht für einen einzelnen untersuchten Build implementiert. Zur Anzeige von Metadaten wie des Buildzeitpunktes oder des Status der Build-Jobs wurden Daten von Travis verwendet. In der Mitte wird die Time to Failure für jede für diesen Build untersuchte Algorithmuskonfiguration dargestellt. Für das abgedruckte Bildschirmfoto wurde die Anzeige auf zwei angezeigte Werte reduziert. Am unteren Ende der Seite werden Informationen über die Ausführung der Builds für den fehlgeschlagenen Build und den erfolgreichen Vorgängerbuild, die zu Studienzwecken wiederholt wurde, gezeigt. Zudem wird die Schnittmenge an fehlschlagenden Tests bei Travis und der wiederholten Buildausführung angegeben.

2. Ausführungsinformationen aus @BeforeClass- und @AfterClass-Methoden wurden nicht aufgezeichnet und auch nicht von der TIA in der Testfallselektion und -priorisierung als Faktor einberechnet.
3. Parametrisierte Tests wurden immer als einzeln ausführbar angenommen.
4. Wurden Tests aus verschiedenen Testklassen priorisiert, so enthielt die vorgeschlagene Testausführungsreihenfolge nicht alle Tests einer Klasse in Folge.

Zu (1): Angenommen, Testfälle hätten das Format *Modul+Testfallnummer*: Wird nicht beachtet, aus welchem Modul ein Test stammt, so kann die geordnete Testausführungsliste diese Teilsequenz von Testfällen enthalten: A+1, B+9, A+3, C+2, A+2. Es werden also Tests hintereinander zur Ausführung vorgeschlagen, die aus unterschiedlichen Modulen stammen, und später wird wieder auf ein bereits vorher begonnenes Modul zurückgekommen. In der heute gängigen Implementation der Unterstützung von JUnit Tests innerhalb von Maven (oder auch Gradle) ist die Ausführung in dieser Reihenfolge nicht möglich. Stattdessen werden Tests modulweise ausgeführt.

Zu (2): Wie in Abschnitt 4.3.1 beschrieben, wurde vor dieser Arbeit lediglich vom Start eines einzelnen Testfalls (Methode oder parametrisierter Test) aufgezeichnet. Coverage, die der Agent möglicherweise in der Zwischenzeit aufnahm, wurde verworfen. In der nun verwendeten Version wurden sowohl der Agent als auch die TIA Implementation in Teamscale um die Unterstützung für @BeforeClass- und @AfterClass-Methoden erweitert. Besonderheiten in der Implementation unter Verwendung von Maven sind im Kapitel 8 beschrieben.

Zu (3): In der Ausführung von parametrisierten Tests mit einzelnen Parametern kann durch geeignete Priorisierung ein Testfehler noch schneller erkannt werden. Die gängigen Testsysteme, die in Builds integriert sind, sehen die Aufforderung zur Ausführung von parametrisierten Tests mit einzelnen Parametern nicht vor.

Zu (4): Testsysteme führen in der Regel alle Testfälle einer Klasse hintereinander aus. Die bisher vorgeschlagene Ausführungsreihenfolge von Testfällen hielt sich nicht an diese Regel. Dies kann insbesondere dann zu Problemen führen, wenn aufgrund von Testabhängigkeiten ein bestimmter Zustand des System unter Tests erwartet wird, der allerdings aufgrund von nur teilweiser Ausführung einer Testklasse nicht besteht.

Für die vier hier beschriebenen Beschränkungen hinsichtlich der Selektion beziehungsweise Priorisierung wurden Optionen (im Folgenden: Realitätsoptionen oder Realitätskonfiguration) in die TIA Implementation von Teamscale eingebaut. Beim Aufruf des entsprechenden Dienstes kann für jede Einschränkung angegeben werden, ob sie gelten soll oder nicht.

Die eingestellte Realitätskonfiguration tritt in dieser Arbeit als Suffix des genutzten Priorisierungsalgorithmus auf. Folgende Konfigurationen fanden Anwendung:

- **CMDP**: Alle Tests einer Klasse müssen hintereinander ausgeführt werden, Tests werden modulweise ausgeführt, @BeforeClass- und @AfterClass-Methoden werden nicht ausgeschlossen, parametrisierte Tests können nicht einzeln ausgeführt werden.
- **MD**: Tests werden modulweise ausgeführt, @BeforeClass- und @AfterClass-Methoden werden nicht ausgeschlossen, parametrisierte Tests können nicht einzeln ausgeführt werden.
- **MP**: Tests werden modulweise ausgeführt, @BeforeClass- und @AfterClass-Methoden werden ausgeschlossen, parametrisierte Tests können nicht einzeln ausgeführt werden.
- **w/o res**: Die Selektion und Priorisierung findet lediglich auf Basis einzelner Tests und ohne Beachtung von Modul- oder Klassengrenzen statt; parametrisierte Tests können mit einzelnen Parametern aufgerufen werden.

8.3.2 Erweiterung des Mutation Testing Benchmarks

Das von Rainer Niedermayr entwickelte und der Forschungsgruppe zugängliche Programm `tia-mutation-benchmark` wurde zur Berechnung relevanter Benchmarks für die mutationsbasierte Analyse verwendet. Dafür waren Anpassungen notwendig, die in den folgenden Absätzen beschrieben werden.

Aufruf der benchmarkrelevanten Algorithmuskonfigurationen

Ursprünglich vergleicht das Werkzeug von Niedermayr einen Algorithmus unter Zuhilfenahme unterschiedlicher Eingabedaten wie Informationen, ob eine Methode scheingetestet oder aufgrund ihrer Trivialität weniger testwürdig ist, und die Stack-Distanz eines Tests zu den getesteten Methoden [NW]. Die Ergebnisse dieser Konfigurationen werden mit einer intern berechneten zufälligen Ausführungsreihenfolge verglichen.

Das Werkzeug wurde so erweitert, dass alle für die vorliegende Studie relevanten Ausführungsreihenfolgen vom Teamscale-Server abgerufen werden.

Außerdem wird die zuvor intern berechnete, zufällige Ausführungsreihenfolge nun auch von Teamscale-Server abgeholt, um Before- und AfterClass-Ausführungsinformationen korrekt zu berücksichtigen.

Erweiterung auf die präzisere Berechnung von Benchmarkwerten für parametrisierte Tests

Die Möglichkeit zur Ausführung eines parametrisierten Tests mit einem bestimmten Parameter ist in der ursprünglichen Umsetzung nicht in der Berechnungslogik implementiert. Es wurde stattdessen angenommen, dass ein parametrisierter Test nur in seiner Gesamtheit, also unter Verwendung sämtlicher Parameter ausgeführt werden kann.

Für die vorliegende Studie, sollte es für die Realitätsoption `mergeParametrizedTests=false` möglich sein, auch die je Parameter getrennte Ausführung von parametrisierten Tests zu emulieren. Aus diesem Grund wurden die entsprechenden Stellen in der Datenstruktur und der Berechnung angepasst.

Möglichkeit der Emulierung paralleler Testausführung je Modul

Für die GitHub-Travis-Projekte wird angenommen, dass die einzelnen Module sequentiell ausgeführt werden. Nutzt man im Benchmark also die Realitätsoption `useModulesToClusterize=true` und der Testfehlschlag tritt im zweiten ausgeführten Modul auf, so ist die gesamte Testausführungszeit (bei Testselektion die entsprechende Ausführungszeit der selektierten Menge) für das vorhergehende Modul in der Zeit-bis-zum-Fehlschlag-Berechnung berücksichtigt.

9 Ergebnisse

In diesem Kapitel werden die Antworten auf die einzelnen Forschungsfragen präsentiert. Es werden zunächst die Ergebnisse aus dem Benchmark mit real auftretenden Testfehlschlägen, dann aus dem Mutation-Based-Benchmark präsentiert und abschließend die Ergebnisse zu den weiteren Fragestellungen.

9.1 Fragestellungen anhand historischer Buildfehlschläge

Die Ergebnisse aus dem Benchmark, der die Testpriorisierungsalgorithmen anhand tatsächlich auf Travis fehlgeschlagener Builds evaluiert, sind in den folgenden Abschnitten dargelegt.

9.1.1 RQ 1.1: Welcher Algorithmus führt durchschnittlich zur kürzesten Time to Failure?

Die Boxplots zeigen für einen untersuchten Algorithmus den Anteil der gesamten Testausführungszeit an, der nötig ist, um – werden die Testfälle in der vorgeschlagenen Reihenfolge ausgeführt – den ersten Testfehlschlag zu erreichen. Hierbei ist zu beachten, dass, wie im bisherigen Verlauf der Arbeit bereits beschrieben, nicht alle Priorisierungsalgorithmen auf alle Builds angewendet werden konnten. Die hier gezeigten Boxplots aggregieren über alle verfügbaren Ergebnisse. Dies bedeutet auch, dass die Boxplots auf einer unterschiedlichen Anzahl von Builds beruhen können.

Abbildung 9.1 zeigt die Boxplots für alle untersuchten GitHub-Travis-Projekte. Die Boxplots für das Studienobjekt Teamscale befinden sich in Abbildung 9.2 und für die Projekte graphhopper und biojava in den Abbildungen 9.5 und 9.8.

Es wurde zusätzlich berechnet, wie viel Zeit durch den Einsatz einer von FRPS (zufällige Reihenfolge) abweichenden Strategie gespart werden kann. In 6% der Fälle verlängerte sich die Dauer bis zum Testfehlschlag und betrug im Maximum den vierfachen Wert im Vergleich zur Time to Failure unter Verwendung von FRPS. In 50% der Fälle war der Fehler jedoch in weniger als 3% der FRPS-Zeit und in über 75% der Builds in maximal 10,43% der FRPS-Zeit erreicht.

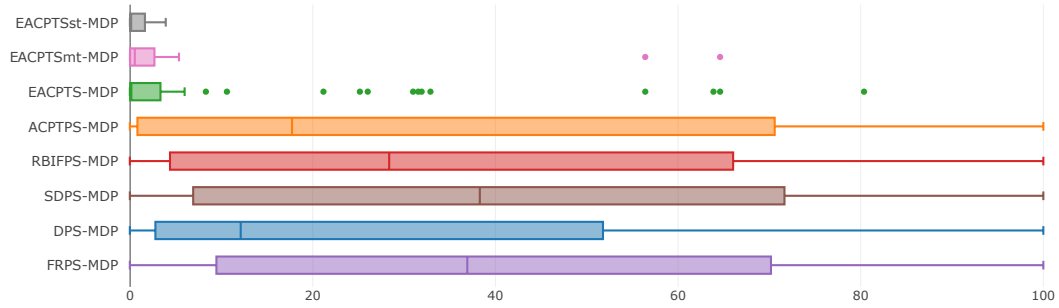


Abbildung 9.1: Benchmark mit realen Testfehlschlägen: Darstellung unterschiedlicher Time to Failure Zeiten über alle analysierten Builds der GitHub-Travis Projekte.

Boxplots für das Projekt teamscale

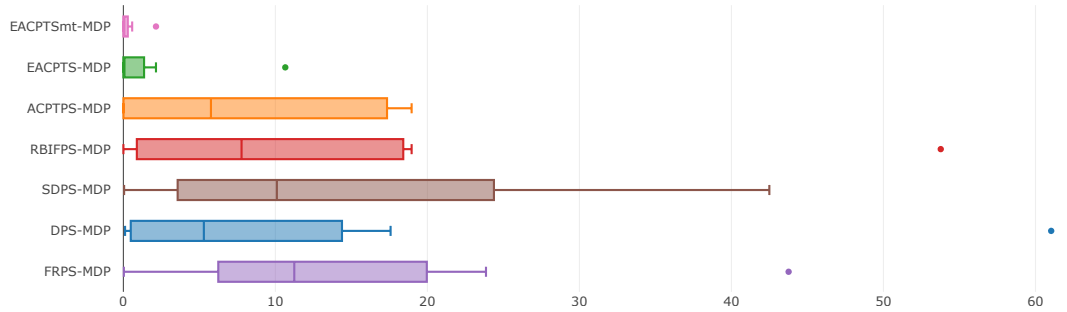


Abbildung 9.2: Benchmark mit realen Testfehlschlägen: Darstellung unterschiedlicher Time to Failure Zeiten über alle analysierten Builds des Projekts *teamscale*.

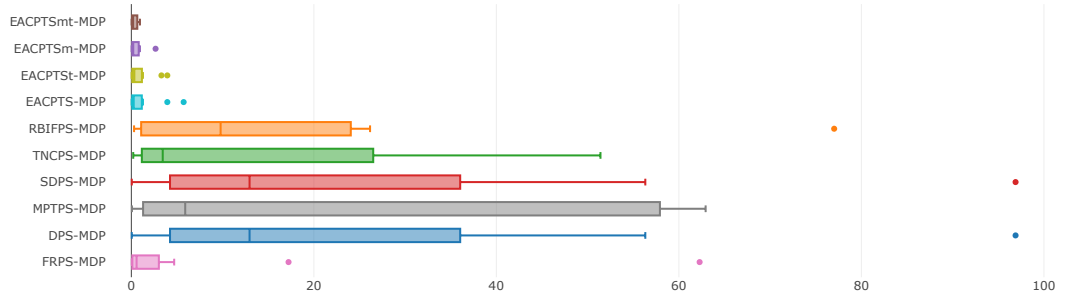


Abbildung 9.3: Mutations-Benchmark: Darstellung unterschiedlicher Time to Failure Zeiten für das Projekt *teamscale* mit fünf zusätzlich als geändert markierten Methoden.

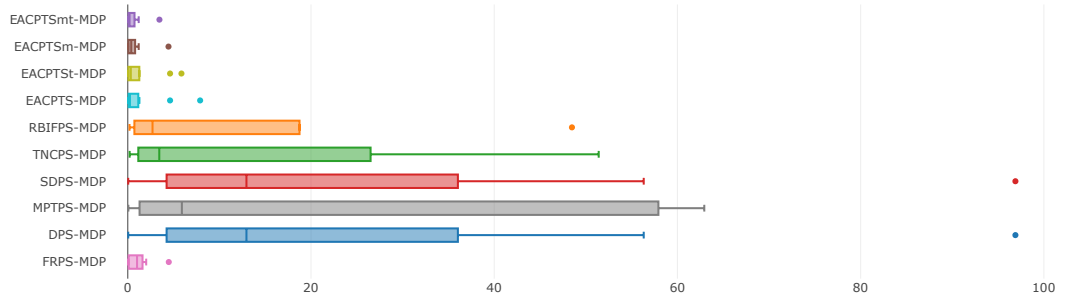


Abbildung 9.4: Mutations-Benchmark: Darstellung unterschiedlicher Time to Failure Zeiten für das Projekt *teamscale* mit zehn zusätzlich als geändert markierten Methoden.

Boxplots für das Projekt graphhopper

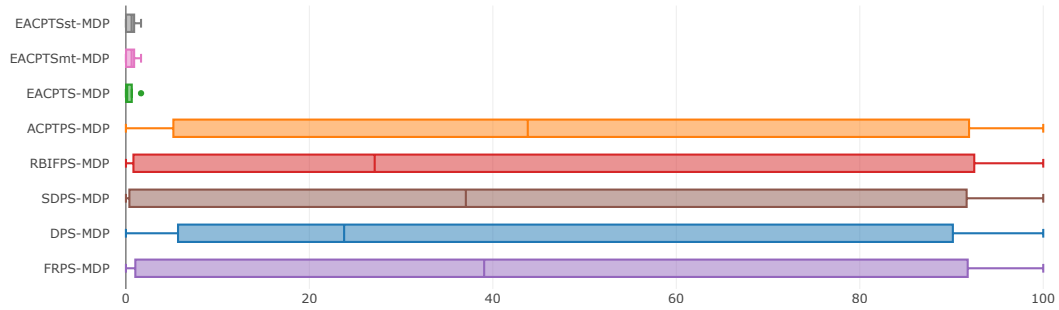


Abbildung 9.5: Benchmark mit realen Testfehlschlägen: Darstellung unterschiedlicher Time to Failure Zeiten über alle analysierten Builds des Projekts *graphhopper*.

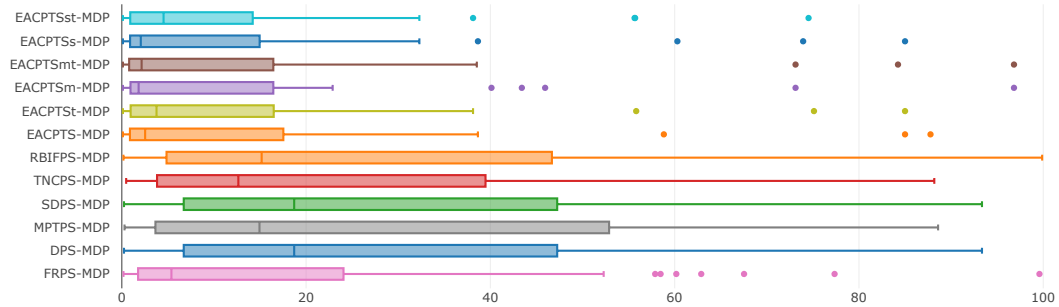


Abbildung 9.6: Mutations-Benchmark: Darstellung unterschiedlicher Time to Failure Zeiten für das Projekt *graphhopper* mit fünf zusätzlich als geändert markierten Methoden.

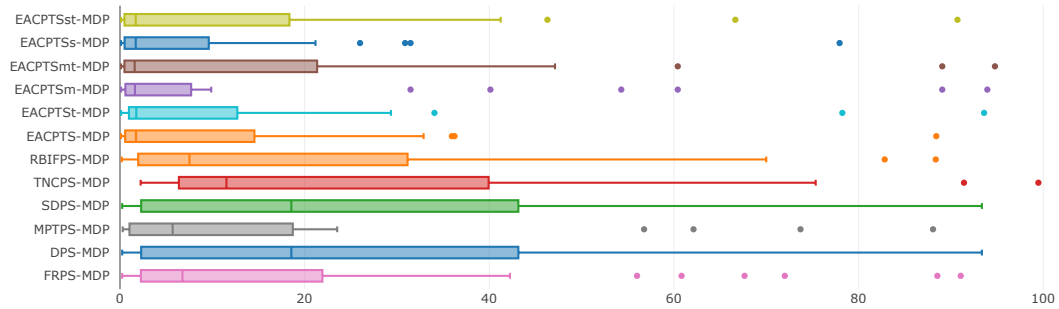


Abbildung 9.7: Mutations-Benchmark: Darstellung unterschiedlicher Time to Failure Zeiten für das Projekt *graphhopper* mit zehn zusätzlich als geändert markierten Methoden.

Boxplots für das Projekt biojava

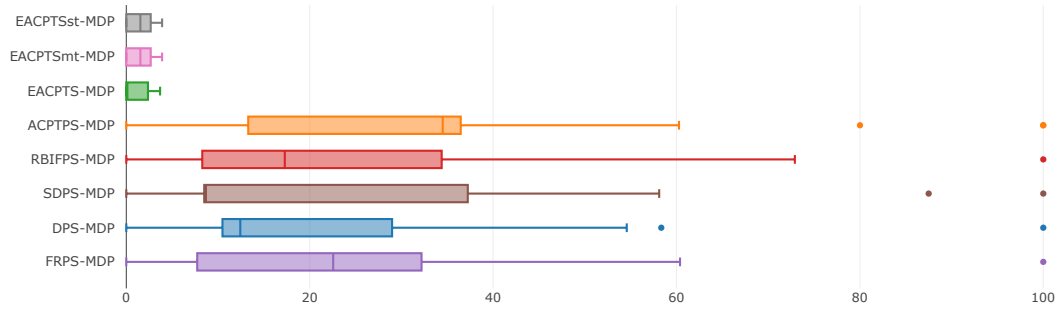


Abbildung 9.8: Benchmark mit realen Testfehlschlägen: Darstellung unterschiedlicher Time to Failure Zeiten über alle analysierten Builds des Projekts *biojava*.

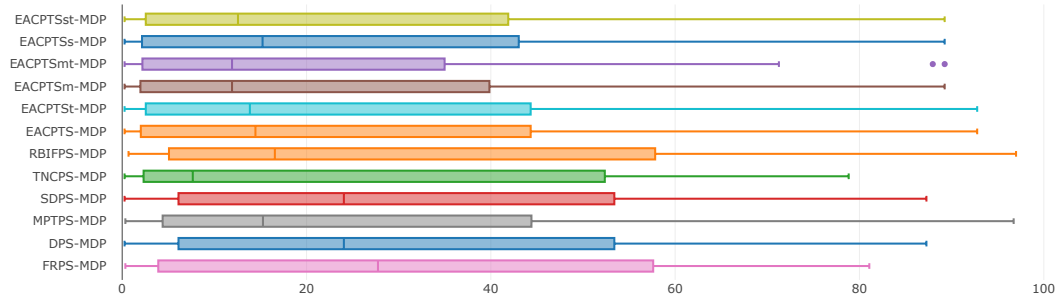


Abbildung 9.9: Mutations-Benchmark: Darstellung unterschiedlicher Time to Failure Zeiten für das Projekt *biojava* mit 5 zusätzlich als geändert markierten Methoden.

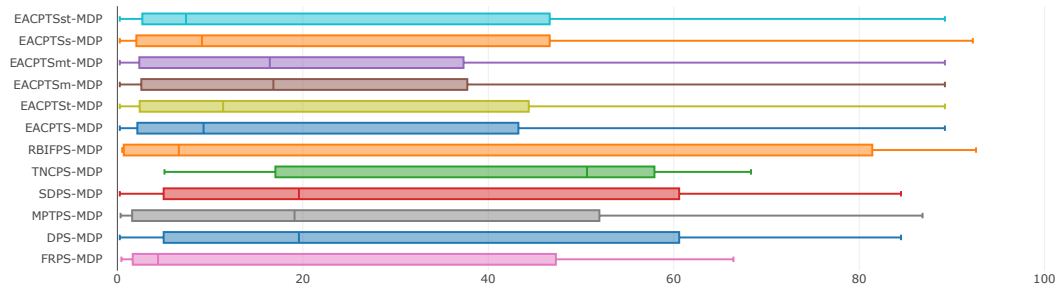


Abbildung 9.10: Mutations-Benchmark: Darstellung unterschiedlicher Time to Failure Zeiten für das Projekt *biojava* mit 10 zusätzlich als geändert markierten Methoden.

Tabelle 9.1: Übersicht über impacted Tests im Benchmark für historische Buildfehlschläge. Die Datenerhebung für diese Forschungsfrage erfolgte nicht über alle Projekte.

Studienobjekt	∅ Anzahl aller Tests	∅ Dauer aller Tests (ms)	∅ Anzahl impacted Tests	∅ Dauer impacted Tests	∅ Anzahl impacted Tests in %	∅ Dauer impacted Tests in %
adamfisk /LittleProxy	181	190115	140	131810	77,35	69,33
apache / commons-lang	3961	46642	11	62	0,28	0,13
biojava / biojava	1099	565520	265	381315	24,11	67,43
bitcoinj / bitcoinj	4701	113902	86	18448	1,83	16,20
doanduyhai / Achilles	550	193269	180	51836	32,73	26,82
dropwizard / dropwizard	1188	210998	175	165422	14,73	78,40
google / gson	1038	6251	7	84	0,67	1,34
graphhopper / graphhopper	1381	190070	375	78696	27,15	41,40
Graylog2 / graylog2-server	1675	149587	35	99336	2,09	66,41
jmxtrans / jmxtrans	296	68330	1	359	0,34	0,53
OfficeDev / ews-java-api	126	1338	2	111	1,59	8,30
openpnp / openpnp	9	23993	7	19746	77,78	82,30
OpenTSDB / opentsdb	387	4463	9	36	2,33	0,81
rackerlabs / blueflood	754	53845	96	21756	12,73	40,40
spullara / mustache.java	218	400759	74	135306	33,94	33,76
square / retrofit	564	44723	252	27242	44,68	60,91
undera / jmeter-plugins	1470	163516	19	813	1,29	0,50
					20,92	35,00

9.1.2 RQ 1.2: Wie hoch ist der Anteil an impacted Tests vom letzten erfolgreichen bis zum fehlschlagenden Build?

Für die untersuchten GitHub-Travis-Projekte ist je die absolute und relative durchschnittliche Anzahl an impacted Tests und Ausführungsdauer in Tabelle 9.1 angegeben.

9.1.3 RQ 1.3: Häufigkeit von unentdeckten Testfehlschlägen durch die Selektierung von Testfällen mit dem vorgeschlagenen Verfahren und optionalen Anpassungen

Die Datenlage ließ die Feststellung nicht aufgedeckter Testfehlschläge durch die Selektion auf Basis von 120 Builds zu, wovon in 2 Fällen der fehlschlagende Test nicht in der selektierten Testausführungsliste enthalten war. Die Überprüfung erfolgte anhand der in der Datenbank persistierten Time to Failure Resultate.

Bei einer Überprüfung der Werte auf abweichendem Wege unter Verwendung der vollständig in der Datenbank abgelegten Testausführungslisten konnten keine zusätzlichen Nichtaufdeckungen festgestellt werden.

9.2 Fragestellungen anhand von Mutationsanalysen

In den folgenden drei Abschnitten werden die Daten, die anhand des Mutation-Based-Benchmarks erhoben wurden, vorgestellt.

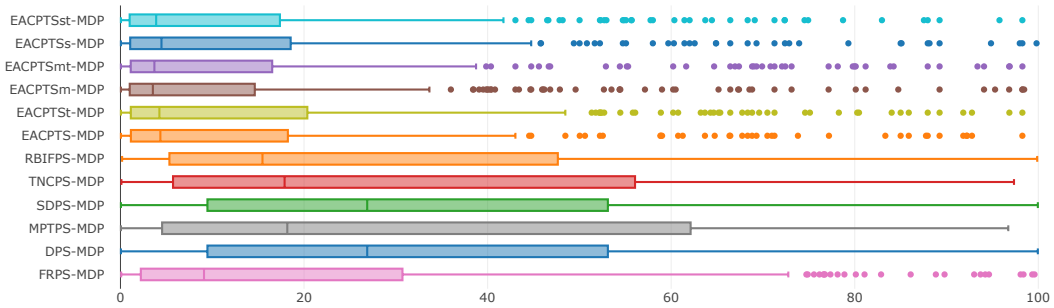


Abbildung 9.11: Darstellung unterschiedlicher Time to Failure Zeiten im mutationsbasierten Benchmark mit 5 zusätzlich als geändert markierten Methoden

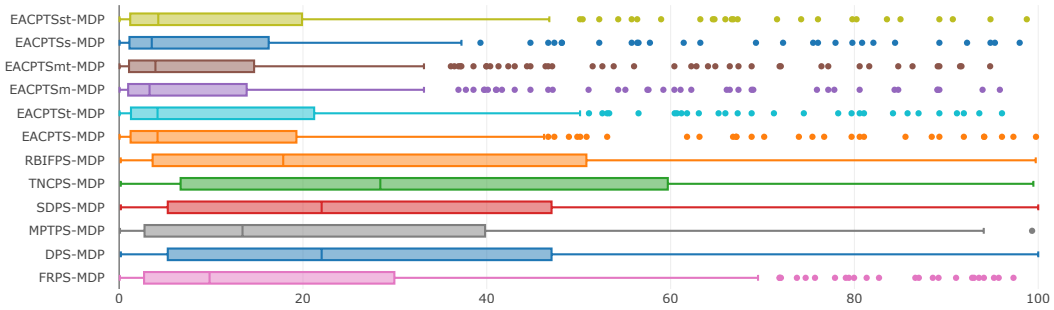


Abbildung 9.12: Darstellung unterschiedlicher Time to Failure Zeiten im mutationsbasierten Benchmark mit 10 zusätzlich als geändert markierten Methoden

9.2.1 RQ 2.1: Welcher Algorithmus führt durchschnittlich zur kürzesten Time to Failure?

Die Abbildungen zur Darstellung der Daten zu dieser Frage folgen dem bekannten Design aus Forschungsfrage RQ 1.1. Die Diagramme sind jeweils getrennt erstellt für die Daten von Berechnungen, in denen 5 Methoden zusätzlich als geändert markiert wurden, und von solchen, in denen 10 Methoden zusätzlich als geändert markiert wurden.

Die Daten über alle untersuchten Projekte finden sich in den Abbildungen 9.11 und 9.12, für das Projekt biojava in Abb. 9.9 und 9.10 sowie für graphhopper in Abb. 9.6 und 9.7. Für das Studienobjekt Teamscale befinden sich die Boxplots in den Abbildungen 9.3 und 9.4.

Für das Studienobjekt Teamscale sind zusätzlich zwei Vergößerungen der Boxplots für die verschiedenen ausgeführten Konfigurationen der EACPTS im Anhang auf Seite 64 abgedruckt.

9.2.2 RQ 2.2: Wie hoch ist der Anteil an impacted Tests durch die Mutationen?

Für die untersuchten Projekte im mutationsbasiertem Benchmark ist je die absolute und relative durchschnittliche Anzahl an impacted Tests und Ausführungsdauer in Tabelle 9.2 angegeben.

Tabelle 9.2: Übersicht über impacted Tests im Mutation-Benchmark. Die Datenerhebung für diese Forschungsfrage erfolgte nicht über alle Projekte.

Studienobjekt	∅ Anzahl aller Tests	∅ Dauer aller Tests (ms)	∅ Anzahl impacted Tests	∅ Dauer impacted Tests	∅ Anzahl impacted Tests in %	∅ Dauer impacted Tests in %
biojava / biojava	1162	763100	99	146184,01	8,56	19,16
bitcoinj / bitcoinj	5296	134749	650	52299,47	12,27	38,81
graphhopper / graphhopper	1700	132576	540	51800,61	31,79	39,07
google / gson	1039	5421	556	3244,35	53,5	59,85
jhy / jsoup	658	11332	468	8878,9	71,05	78,35
adamfisk / LittleProxy	186	154820	173	148093,41	93,22	95,66
traccar / traccar	302	3898	42	533,88	13,95	13,7
apache / commons-lang	4057	41153	74	976,99	1,82	2,37
Esri / geometry-api-java	407	8720	171	4565,57	42,07	52,36
dropwizard / dropwizard	1650	156875	195	40311,75	11,81	25,7
qos-ch / logback	1013	26055	136	5788,02	13,38	22,21
square / retrofit	597	15580	179	6569,23	29,98	42,16
					31,95	40,78

9.2.3 RQ 2.3: Häufigkeit von unentdeckten Testfehlschlägen durch die Selektierung von Testfällen mit dem an der Forschungsgruppe vorgeschlagenen Verfahren

In den neun untersuchten Projekten, von denen je 150 zufällig ausgewählte Mutationen im Benchmark untersucht wurden, konnten in zwei Projekten insgesamt 15 Mutationen bei der Anwendung des an der Forschungsgruppe entwickelten Selektionsverfahrens nicht mehr aufgedeckt werden. Dies war der Fall im Projekt biojava: je 2 Mutanten im Benchmark mit 5 (100 ausgewählte Mutanten) und 10 (50 ausgewählte Mutanten) zusätzlich dem Code-Delta hinzugefügten Methoden. Und im Projekt retrofit: 7 Mutanten im Benchmark mit 5 zusätzlich als geändert ausgewählten Methoden und 4 Mutanten in der Konfiguration mit 10 zusätzlich als geändert ausgewählten Methoden.

9.3 Fragestellungen zu technischen Einschränkungen im Testverfahren und möglicher Vereinfachung im Prozess

Die Ergebnisse zum dritten Fragenkomplex, in dem untersucht wurde, welchen Einfluss verschiedene Realitätskonfigurationen oder ältere Testausführungsdaten auf die Time to Failure haben und wie häufig @BeforeClass- und @AfterClass-Methoden in den untersuchten Projekten eingesetzt wurden, sind im Folgenden wiedergegeben.

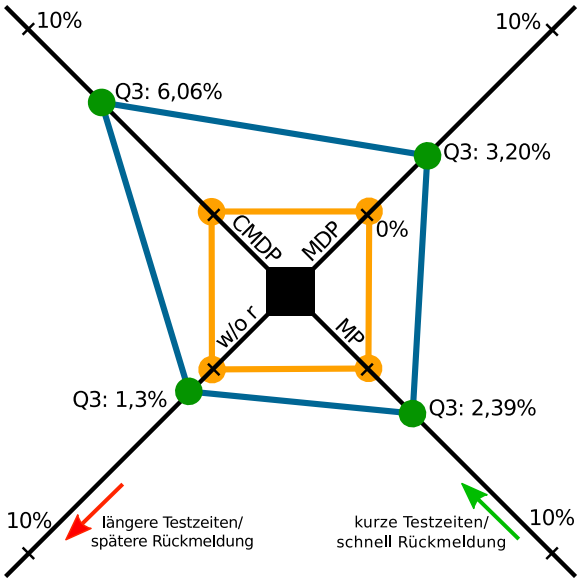


Abbildung 9.13: In dieser Darstellung sind Median (orange) und 3. Quartil (blau-grün) der durchschnittlichen Time to Failure Zeiten über alle untersuchten Projekte gezeigt. Dies ermöglicht einen einfachen Vergleich der Werte über verschiedene Realitätsoptionen hinweg. Auf jeder der sternförmig nach außen gerichteten Achsen sind die Werte für eine Realitätskonfiguration abgebildet; Metrik ist die benötigte relative Testzeit in Prozent. Die zu erwartende Zeit bis zum Fehlschlag ist höher, wenn sich die Punkte weiter außen befinden.

9.3.1 RQ 3.1: Wie wirken sich Beschränkungen in der Ausführungsreihenfolge von Tests (z. B. innerhalb von Modulen oder Klassen) auf die Time to Failure aus?

In Abbildung 9.13 wird die Performanz hinsichtlich der Zeit bis zum ersten erreichten Testfehlschlag unter Verwendung unterschiedlicher Realitätskonfigurationen gezeigt.

Ein anderes Bild zeigt sich, wenn der Algorithmus ohne vorherige Testselektion, also als reiner Priorisierungalgorithmus eingesetzt wird. Die Daten hierfür repräsentieren die Boxplots in Abbildung 9.14.

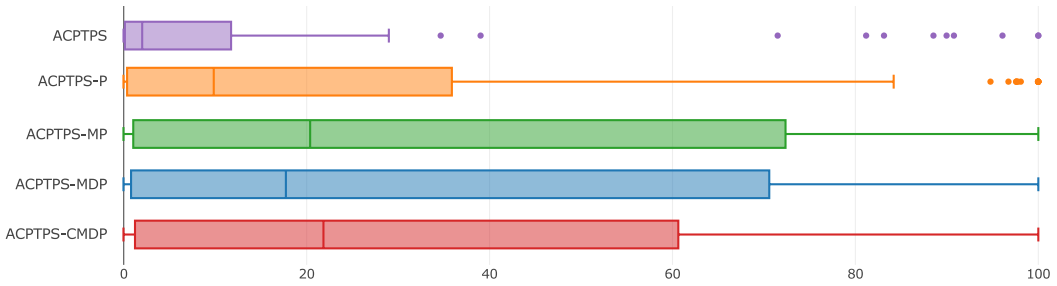


Abbildung 9.14: Wird lediglich der Priorisierungsanteil des an der Forschungsgruppe vorgeschlagenen Priorisierungs- und Selektionsverfahrens verwendet, so zeigen sich stärkere Effekte der Realitätskonfigurationen.

Tabelle 9.3: Übersicht über aufgezeichnete Ausführungszeiten und Häufigkeiten von @BeforeClass- und @AfterClass-Methoden. Der Übersichtlichkeit halber sind Nullwerte nicht abgedruckt.

Studienobjekt	∅ Gesamttestdauer (s)	∅ Anzahl before	∅ Dauer (s) before	∅ Anzahl inbetween	∅ Dauer (s) inbetween	∅ Anzahl after	∅ Dauer (s) after	∅ Anteil Testdauer	Anzahl Builds
biojava / biojava	553			2	0,219			0,03%	24
doanduyhai / Achilles	266	6	96,587	31	11,930			40,80%	11
dropwizard / dropwizard	222	1	< 0,001	9	0,934			0,45%	3
graphhopper / graphhopper	200	2	3,383	14	2,654			3,01%	10
Graylog2 / graylog2-server	272			8	0,534			0,19%	3
jamesagnew / hapi-fhir	356	8	4,672	67	10,214			4,18%	3
spullara / mustache.java	400			1	0,006			<0,01%	1

9.3.2 RQ 3.2: Wie häufig und langandauernd sind klasseneigene Setup- und Teardown-Methoden?

Es wurden die Daten von GitHub-Travis-Projekten aus dem Benchmark mit real fehlschlagenden Testfällen verwendet. Dabei konnte für sieben von 23 Projekten eine Verwendung von klassenspezifischen Setup- und Teardown-Methoden festgestellt werden.

Die aufgezeichneten Daten zur Häufigkeit und Ausführungsdauer von @BeforeClass- und @AfterClass-Methoden sind in Tabelle 9.3 abgedruckt. In dieser Tabelle finden sich ebenfalls Daten zur Frage, wie häufig Code-Ausführungen zwischen zwei Testklassen festgestellt wurden, die aufgrund der in Abschnitt 8.1 beschriebenen technischen Gegebenheiten weder der zuletzt ausgeführten Klasse noch der danach ausgeführten Klasse eindeutig zugeordnet werden konnte.

9.3.3 RQ 3.3: Was sind die Auswirkungen einer unregelmäßigen Aktualisierung von Test-zu-Methoden-Beziehungen?

Die Berechnungen der Time to Failure unter Verwendung der EACPTS wurden mit alten Testausführungsinformationen für das Projekt *graphhopper* wiederholt. In Abbildung 9.15 sind die absoluten Zeiten bis zum Testfehlschlag in einem gruppierten Balkendiagramm dargestellt. Der linke, blaue Balken der Gruppe stellt die Time to Failure aus RQ 1.1 dar (Testausführungsinformationen des grünen Vorgängercommits werden genutzt). Der rechte, orange Balken repräsentiert die Zeiten unter Verwendung der älteren Testausführungsdaten. In Abbildung 9.16 sind die relativen Werte abgedruckt. Von links nach rechts steigt im Diagramm die Zeitspanne zwischen genutzter Coverage und fehlgeschlagenem Build.

Rot markiert sind die Commits, für die in der neuen Konfiguration der fehlschlagende Test nicht mehr ausgeführt wurde. Dies traf in fünf von 15 Fällen zu.

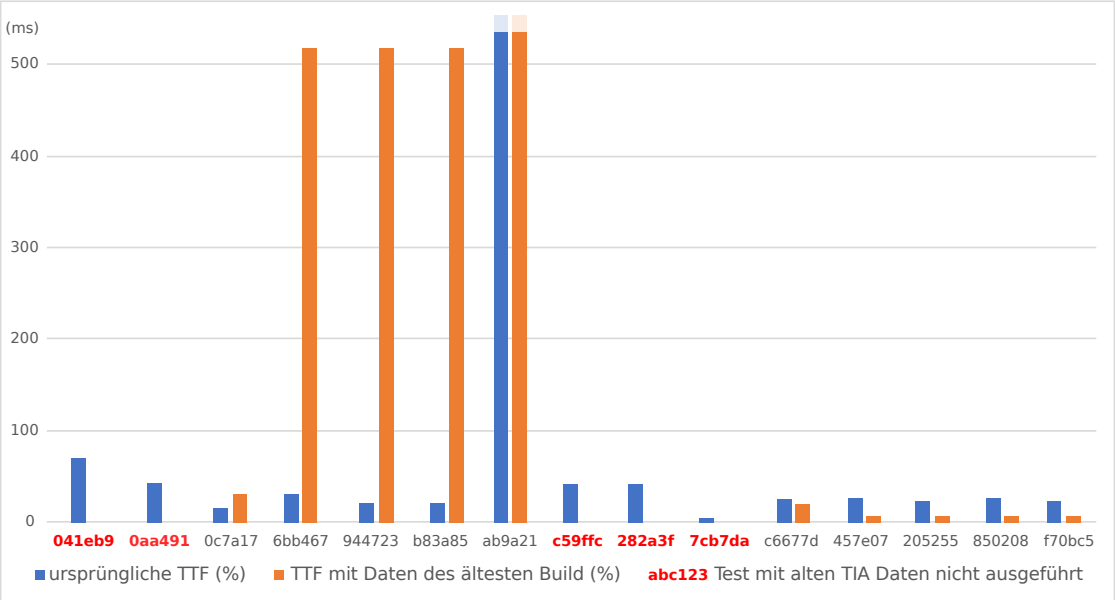


Abbildung 9.15: Absolute Time To Failure Zeiten bei der Nutzung alter Testausführungsdaten (orange Balken) im Vergleich zu den Werten (blaue Balken) aus Forschungsfrage RQ 1.1, in der aktuelle Testausführungsdaten genutzt wurden.

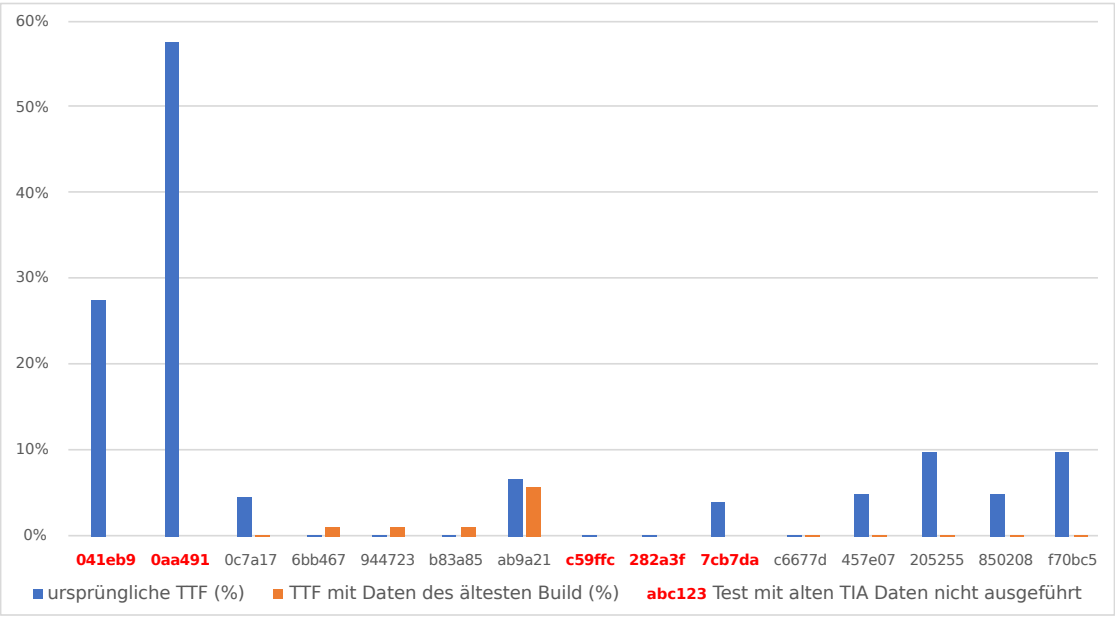


Abbildung 9.16: Relative Time To Failure Zeiten bei der Nutzung alter Testausführungsdaten (orange Balken) im Vergleich zu den Werten (blaue Balken) aus Forschungsfrage RQ 1.1, in der aktuelle Testausführungsdaten genutzt wurden.

10 Diskussion

In den folgenden Abschnitten sollen die Ergebnisse aus dem vorherigen Kapitel ausgewertet werden. Dabei wird zunächst auf die Forschungsfragen eingegangen. Im späteren Teil des Kapitels werden die Ergebnisse von drei Projekten im Benchmark bei real auftretenden Testfehlschlägen mit den Ergebnissen im Mutation-Based-Benchmark verglichen. Außerdem wird auf generelle Aspekte der Studie und auf mögliche Einschränkungen der Validität eingegangen.

10.1 Fragestellungen anhand historischer Buildfehlschläge

Die Fragestellungen, die anhand des Benchmarks mit real fehlgeschlagenen Tests geklärt werden sollen, werden als für den Alltag von Softwareentwicklern besonders relevant eingeschätzt. Wie im Verlauf der Arbeit bereits erläutert, kann eine schnelle Rückmeldung von CI-Server die Produktivität des Entwicklers fördern und ist außerdem hilfreich, um einen neu aufgetretenen Fehler im Softwaresystem frühzeitig zu beseitigen. Dies kommt der Qualität des zu entwickelnden Systems zugute und trägt zu weniger Feldfehlern bei.

10.1.1 RQ 1.1: Welcher Algorithmus führt durchschnittlich zur kürzesten Time to Failure?

Die Analyse des Boxplots über alle Builds sowie die Werte der Mediane und der oberen Quartile geben ein deutliches Bild hinsichtlich der Vorteile einer Testfallselektion ab.

Das dritte Quartil der EACPTS liegt in allen Variationen unter 5% der gesamten Testausführungszeit. Im Vergleich zu SDPS, FRPS und RBIF liegt das dritte Quartil der EACPTS sogar vor dem ersten Quartil der anderen. Das bedeutet, in der Zeit, in der in 75% der Fälle der Testfehlschlag unter Verwendung der EACPTS erreicht wurde, war dies in unter 25% der Fälle für die anderen Algorithmen der Fall. Im Durchschnitt benötigte die Berechnung der Testausführungsliste etwa 13 Sekunden.

Bei den reinen Priorisierungsalgorithmen schneidet die SDPS am schlechtesten ab. Median und drittes Quartil liegen jeweils knapp hinter den Werten für die FRPS. Über das erste Viertel der Builds betrachtet erreicht die SDPS zwar den Testfehler etwas früher (in 7% der relativen Testausführungszeit im Vergleich zu 10% bei der FRPS). Dies rechtfertigt allerdings nicht den sehr viel höheren Zeitaufwand in der Berechnung (im Durchschnitt 67 Sekunden pro Priorisierung statt 1,6 Sekunden für die Ausgabe der randomisierten Testliste).

Durch die ACPTPS ließen sich in 25% der Fälle ein Testfehler sehr früh aufdecken (< 2% der relativen Testausführungszeit). In 50 der untersuchten Builds wurde der fehlschlagende Test in unter 18% der gesamten Testasuführungszeit erreicht. Dieser Wert ist deutlich schlechter, wenn man die Ergebnisse bei über 75% der Builds betrachtet. Das dritte Quartil liegt mit 72% etwa gleich auf mit denen von der SDPS und FRPS. Die gemessene Berechnungszeit für die ACPTS war mit unter 3 Sekunden gering.

In 50% der Fälle liefert auch der Priorisierungsalgorithmus DPS, der die Tests nach der letztmaligen Ausführungsdauer sortiert, mit 12% der Testausführungsdauer ein gutes

Ergebnis. Insgesamt liegen bei der DPS Median und drittes Quartil am niedrigesten, beschränkt man die Betrachtung auf die reinen Testpriorisierer. Aufgrund der vernachlässigbar kurzen Sortierzeit verspricht die DPS, in vielen Projekten gute Ergebnisse zu erzielen ohne Tests zu selektieren.

Etwas überraschend mögen die Zeiten des RBIFPS sein. Obwohl das erste Quartil mit unter 7% einen guten Wert aufweist, liegt dieser auffällig hinter zum Beispiel dem der ACPTS. Auf die zugrundeliegende Buildmenge bezogen, spricht dieses Ergebnis dafür, dass sich durch die Sortierung von Tests nach ihrer Ausführunggeschwindigkeit bessere Priorisierungserfolge erzielen lassen als allein durch die Platzierung von impacted Tests an den Anfang der Testausführungliste.

Betrachtet man die Grafik, die sich auf Builds des Projekts biojava beschränkt, so weicht das Ergebnis nicht nur wenig von dem über alle Builds hinweg betrachteten ab. Beim Projekt graphhopper liegen die ersten Quartile für die anderen Algorithmen auch bei sehr niedrigen Werten. Dafür sind die dritten Quartile bei mehr als 90% der gesamten Testausführungszeit. Die EACPTS dagegen erreicht in diesem Projekt den fehlschlagenden Test in allen Fällen in unter 3% der Testausführungszeit.

10.1.2 RQ 1.2: Wie hoch ist der Anteil an impacted Tests vom letzten erfolgreichen bis zum fehlschlagenden Build?

Der prozentuale Anteil an impacted Tests schwankt von Projekt von Projekt stark. Dennoch werden dem Benchmark zufolge respektable Einsparungen hinsichtlich der Ausführungszeit erreicht. So werden im Durchschnitt in einem Drittel (35%) der gesamten Testausführungszeit alle impacted Tests ausgeführt.

Dies zeigt, dass sich ein Code-Delta mit deutlich weniger Tests (etwa 20%) als der gesamten Testsuite abdecken lässt und damit eine entsprechende Zeitersparnis einhergeht. In fünf von 17 untersuchten Projekten lag der durchschnittliche Zeitaufwand zur Ausführung der impacted Tests bei unter 1,5% der gesamten Testdauer.

Explizit sei an dieser Stelle erwähnt, dass die Time to Failure nach der Selektion maximal die Dauer zur Ausführung aller impacted Tests annehmen kann, allerdings durch die Priorisierung in vielen Fällen deutlich geringere Werte möglich sind. Dies zeigte sich

Unabhängig vom konkreten Wert der unentdeckten Testfehlschläge ist in jedem Fall zu raten, regelmäßig die komplette Testsuite auszuführen, um auch jene Tests zur Ausführung kommen zu lassen, die nicht durch die Selektion bereits in einem Build ausgeführt wurden. Bei sehr lang andauernden oder teuren Tests muss gegebenenfalls das Risko abgeschätzt werden, das mit einem langen Intervall ohne vollständige Testung verbunden ist.

10.1.3 RQ 1.3: Häufigkeit von unentdeckten Testfehlschlägen durch die Selektierung von Testfällen mit dem vorgeschlagenen Verfahren und optionalen Anpassungen

Das Nichtausführen von Tests, die aufgrund von Regressionen fehlschlügen, bringt eine Gefährung der Softwarequalität mit sich. Es erhöht das Risiko, dass Fehler lange unentdeckt bleiben und im schlimmsten Fall als Feldfehler in Erscheinung treten.

Dass auf Basis von 120 Builds lediglich zwei Testfehler nicht aufgedeckt wurden, kann als positiv bewertet werden. Anzumerken ist hierbei allerdings, dass einige der untersuchten Builds (vgl. Abschnitt 7.1.3) aus der Vorgängerstudie [Rot18] stammen und von dort die Builds ausgewählt wurden, deren Testfehlschläge nach der Selektion aufgedeckt wurden. Außerdem konnten aufgrund der Datenlage (siehe Abschnitt 10.6) nicht alle Builds, auf

deren Basis zum Beispiel die Boxplots erzeugt wurden, für diese Berechnung in Betracht gezogen werden. Es wurden nur diejenigen Builds verwendet, für die sowohl Daten für die EACPTS als auch die DPS vorlagen.

10.2 Fragestellungen anhand von Mutationsanalysen

In den folgenden drei Abschnitten werden die Fragestellungen zum Mutations-Benchmark diskutiert.

10.2.1 RQ 2.1: Welcher Algorithmus führt durchschnittlich zur kürzesten Time to Failure?

Es wird zuerst Abbildung 9.11 mit den Ergebnissen über alle Builds und unter Verwendung des um fünf Methoden erweiterten Code-Deltas betrachtet. Anschließend werden eventuelle Auffälligkeiten im Vergleich zu den um zehn Methoden erweiterten Code-Deltas besprochen.

Alle Strategien erreichen in 75% der Fälle den fehlschlagenden Test in maximal 63% der gesamten Testausführungszeit. Generell fallen in den Time to Failures der EACPTS viele Ausreißer auf. Ansonsten liegt das dritte Quartil für alle Variationen dieser Strategie bei unter 21% der gesamten Testausführungszeit und damit deutlich vor den Median-Werten von DPS und SDPS. Die Strategien TNCPS und RBIF haben Mediane von zwischen 16% und 18%, allerdings dritte Quartile zwischen 48% und 55% und performen also deutlich schlechter.

Dass die besten Time to Failure Werte die EACPTS liefert, ist durch die Kombination von Selektions- und Priorisierungsmechanismus zu erklären. Die Zeit des Verfahrens RBIF-PS, das impacted Tests an den Anfang der Ausführungsliste stellt, aber innerhalb der beiden Gruppen *impacted* und *non-impacted* in einer zufälligen Reihenfolge anordnet, ist den Messungen zufolge weniger effektiv.

Starke Unterschiede in den unterschiedlichen Konfigurationen zum erweiterten Code-Delta können nicht festgestellt werden. Jedoch ist das dritte Quartil der MPTPS bei 10 zusätzlich geänderten Methoden deutlich geringer (40% statt 63%). Ebenfalls leicht besser sind die dritten Quartile von DPS und SDPS, wohingegen sich die von TNCPS und RBIFPS geringfügig verschlechtern.

10.2.2 RQ 2.2: Wie hoch ist der Anteil an impacted Tests durch die Mutationen?

Die Menge der impacted Tests ist im Mutations-Benchmark im Schnitt rund 32 Tests groß, und die Ausführung derselben dauert im Mittel 40,78% der gesamten Testzeit.

Wie auch beim Benchmark mit realen Testfehlschlägen zeigt sich im Vergleich mit den Boxplots zur EACPTS, dass der erste Testfehlschlag meist deutlich früher entdeckt wird, nämlich in 75% der Fälle nach nicht einmal der Hälfte der Ausführungszeit für die impacted Tests. Die angewendete Priorisierung des Algorithmus ist neben der Selektion also ein wichtiger Faktor für das schnelle Erreichen eines Testfehlschlags.

10.2.3 RQ 2.3: Häufigkeit von unentdeckten Testfehlschlägen durch die Selektierung von Testfällen mit dem an der Forschungsgruppe vorgeschlagenen Verfahren

Die nicht aufgedeckten 15 Mutanten machen bei den 1350 untersuchten Mutationen einen Anteil von 1,11% aus. Wie in der Interpretation zu Forschungsfrage RQ 1.2 bereits beschrieben, sind nicht aufgedeckte Fehlschläge problematisch für die Codequalität.

Der hier gemessene Wert von 1,1% der untersuchten Mutationen ist als gering einzustufen. Auffällig ist, dass alle nicht entdeckten Mutationen aus nur zwei (von neun) Projekten stammen. Eine Detailuntersuchung der Hintergründe war im Rahmen dieser Arbeit nicht möglich.

Wie ebenfalls bereits zu Forschungsfrage RQ 1.2 dargelegt, ist das regelmäßige Ausführen – wo möglich – aller Tests zusätzlich zur TIA ratsam. Schnelle Time to Failure-Zeiten sind für den Entwickler gut, allerdings dürfen darunter nicht die Projektqualität leiden und eventuelle Fehler unentdeckt bleiben.

10.3 Vergleich der Time-to-first-Failure für historische Testfehlschläge und Mutationsanalysen anhand der Projekte biojava, graphhopper und Teamscale

Bevor die Ergebnisse des dritten Forschungsfragenkomplexes besprochen werden, wird in diesem Abschnitt ein knapper Vergleich der Performanzen der Priorisierungsalgorithmen gezogen. Es werden die Daten aus dem Benchmark mit realen Testfehlern den Daten aus dem mutationsbasiertem Benchmark gegenübergestellt. Dies geschieht für die Studienobjekte *biojava*, *graphhopper* und *Teamscale*.

Der Fokus des Vergleichs liegt dabei auf dem von der Forschungsgruppe vorgestellen und weiterentwickelten Verfahren EACPTS (Enhanced Additional Coverage Per Time Strategy).

Teamscale Alle Boxplots für das Studienobjekt *Teamscale* befinden sich auf Seite 39. In allen drei Boxplots lässt sich feststellen, dass die EACPTS die beste Performanz hat. Unter den verschiedenen Variationen der EACPTS ist außerdem zu erkennen, dass durch die Verwendung von Daten zur Scheintestung und Trivialität von Methoden ein Zeitgeweinn zu erreichen ist.

Die generell beobachtete maximale Time to Failure ist beim Mutations-Benchmark länger. Dafür ist in diesem die Ausführungsdauer mit der zufälligen Testausführung sehr viel schneller. Da in Teamscale nicht in allen Testbereichen Mutationen hineinmutiert werden konnten, kann es sein, dass die im Benchmark überprüften Mutanten vor allem in Methoden erzeugt wurden, die durch schnelle Unit-Tests abdeckt werden. Im Benchmark mit den realen Testfehlschlägen traten Fehler in unterschiedlichen Teststufen auf.

graphhopper Alle Boxplots für das Studienobjekt *graphhopper* befinden sich auf Seite 40. Auch in diesem Projekt ist festzustellen, dass die schnellsten Time to Failure-Werte durch den Einsatz von EACPTS erreicht werden. Auffällig hierbei ist, dass im Benchmark mit den real auftretenden Testfehlern alle nicht selektierenden Algorithmen eine sehr große Spannweite haben hinsichtlich der Zeit, die es braucht, um den ersten Testfehler zu erreichen. Dies ist im Mutations-Benchmark nicht zu beobachten.

Dafür liegen die Werte der EACPTS beim Mutations-Benchmark deutlich höher als beim Benchmark mit real auftretenden Testfehlschlägen (drittes Quartil < 17% statt < 3%).

Das Projekt graphhopper spiegelt den allgemeinen Vergleich zwischen den Daten aus dem Mutations-Benchmark und den Daten aus dem Benchmark mit realen Testfehlern am besten wider.

biojava Alle Boxplots für das Studienobjekt *biojava* befinden sich auf Seite 41. Bei diesem Projekt ist der zu beobachtende Unterschied zwischen den Ergebnissen aus dem Benchmark mit real auftretenden Fehlschlägen und dem Mutation-Benchmark am auffälligsten.

Die Boxplots für die Ergebnisse vom erstgenannten Benchmark (Abb. 9.8) zeigen eine „gewohnt" gute Leistung der EACPTS. Im Mutations-Benchmark dagegen sind alle verwendeten Priorisierungsalgorithmen ähnlich leistungsstark.

Vergleicht man die Daten mit den Daten für biojava in den Tabellen 9.1 und 9.2, ist zu erkennen, dass die durchschnittliche Ausführungsdauer aller impacted Tests beim Benchmark mit realen Testfehlschlägen größer ist (Impacted Tests: 24,11%, Dauer: 67,43%) als beim Mutations-Benchmark (Impacted Tests: 8,56%, Dauer: 19,16%). Das lässt darauf schließen, dass die schnellen Funde im Benchmark mit den realen Testfehlschlägen von einer nützlichen Priorisierung herrühren und nicht (primär) Effekte der Selektion sind.

10.4 Fragestellungen zu technischen Einschränkungen im Testverfahren und möglicher Vereinfachung im Prozess

Die in dieser Studie ermittelten Time to Failure-Werte wurden theoretisch berechnet und nicht bei einer erneuten Testausführung aufgezeichnet. Um die Ergebnisse möglichst realitätsnah zu halten, wurde in der Berechnung der Ergebnisse für die Forschungsfragen RQ 1.x und RQ 2.x die Realitätskonfiguration MDP genutzt (vgl. 8.3.1). Im folgenden Abschnitt wird zunächst betrachtet, wie groß der Effekt der unterschiedlichen Realitätsoptionen auf die Time to Failure der EACPTS ist. Danach werden die Daten zur Häufigkeit und Dauer von Setup- und Teardown-Methoden interpretiert und zuletzt die Nutzung der TIA mit älteren Coveragedaten besprochen.

10.4.1 RQ 3.1: Wie wirken sich Beschränkungen in der Ausführungsreihenfolge von Tests (z. B. innerhalb von Modulen oder Klassen) auf die Time to Failure aus?

In der grafischen Darstellung in Abbildung 9.13 werden die Effekte der Realitätsoptionen erkennbar. Generell wird durch die Anwendung der EACPTS eine geringe Time to Failure erzielt. Je restriktiver die Realitätsoptionen werden, desto länger dauert es, bis der Testfehlschlag erreicht wird. Die Darstellung startet mit der nach links oben zeigenden Achse, auf der die Daten der restriktivsten Realitätskonfiguration CMDP liegen und lässt auf den weiteren Achsen im Uhrzeigersinn immer mehr Freiheiten in der Ausführungsreihenfolge zu.

Während das dritte Quartil bei der Realitätsoption ohen Einschränkungen bei 1,3% der relativen Gesamttestdauer liegt, ist der Wert für das dritte Quartil in der Standardkonfiguration MDP 3,2%. Diese Daten unterstützen die Annahme, dass die Beschränkungen in der Testausführungsreihenfolge von der theoretischen Time to Failure-Berechnung des Benchmarksystems beachtet werden sollten.

Ob sich eine Weiterentwicklung der Testausführungssysteme lohnt, sodass die Ausführung zum Beispiel in der Konfiguration MP möglich wird, sollte in einer Aufwands-Nutzen-Analyse ermittelt werden. Die hier festgestellten geringen Unterschiede im Ver-

gleich mit dem derzeit in der Industrie noch zu beobachtenden Testverfahren sprechen eher dafür, den Einsatz von Selektions- und Priorisierungsverfahren in einer einfach umzusetzenden Variante einzuführen. Denn die Werte aus den vorherigen Forschungsfragen legen nahe, dass sich mit diesen Verfahren Testzeit und Ressourcen in hinreichendem Ausmaß sparen lassen.

Soll jedoch in einem Projekt keine Testfallselektion stattfinden, sondern nur priorisiert werden, so spielen die technischen Beschränkungen des Testausführungssystems eine weitaus größere Rolle, wie die Boxplots in Abbildung 9.14 zeigen.

Ist eine beliebige Ausführungsreihenfolge möglich und werden Testabhängigkeiten nicht berücksichtigt, so lassen sich durch die reine Priorisierung sehr gute Time to Failure-Werte erreichen. Diese Werte nehmen bereits zu, wenn man parametrisierte Tests nicht je Parameter ausführen kann, sondern nur alle Ausführungsvarianten des Tests in Folge (Suffix *-P*). Die Zeit bis zum Fehlschlag wird weiterhin länger, wenn man Tests nicht modulübergreifend anordnen kann, unter der gleichzeitigen – hier getroffenen – Annahme, dass Module nicht parallel, sondern sequentiell ausgeführt werden. ACPTPS-MDP zeigt im Vergleich zu ACPTPS-MP bessere relative Ausführungszeiten. Dies ist plausibel, da im Modus MP die @BeforeClass- und @AfterClass-Methoden nicht in die Zeitberechnung einfließen (auch nicht in die Gesamtdauer). Die zusätzliche Bedingung, dass alle Testfälle einer Klasse sequentiell hintereinander ausgeführt werden müssen, zeigt – den Median betrachtet – einen leicht negativen Effekt. Überraschender ist der positive Effekt auf das dritte Quartil. Dieser lässt sich durch die rundenbasierte Priorisierung der Tests klären. Wie in Abschnitt 7.1.1 (S. 23f.) erklärt, verfolgt die ACPTPS das Ziel in jeder Runde durch die Ausführung möglichst schneller Tests einen großen Teil des Systems abzudecken. Ein positiver Effekt der Realitätsoption '-C' kann dann eintreten, wenn zwei Tests einer Klasse den gleichen Teil des Systems abdecken. Ist einer der beiden Tests schneller, so wird dieser der Testausführungsliste hinzugefügt. Der andere dagegen wird für die aktuelle Runde zurückgestellt, da er keine zusätzlichen Teile des Systems abdeckt. Deckt der früher priorisierte Test im Gegensatz zum anderen Test allerdings keinen Fehler auf, so ist die Time to Failure höher, als wäre der zweite Test gleich nach dem ersten Test ausgeführt worden. Diese Anordnung wird durch die Realitätsoption '-C' erreicht. Im Generellen kann aber nicht davon ausgegangen werden, dass sich Testzeiten durch das direkt aufeinander folgende Ausführen aller Test einer Klasse senken lassen.

10.4.2 RQ 3.2: Wie häufig und langandauernd sind klasseneigene Setup- und Teardown-Methoden?

Die aufgezeichneten und in Tabelle 9.3 dargelegten Daten zu @BeforeClass- und @AfterClass-Methoden lassen eine geringe Nutzung dieser klassenspezifischer Setup- und Teardown-Methoden annehmen. Der hohe relative Anteil solcher Methoden an der Testausführungszeit des Projektes Achilles lassen aber auch starke projektspezifische Unterschiede vermuten. In der hier vorliegenden Arbeit ließen sich keine starken Effekte beobachten, was auch die Unterschiede in den Realitätsoptionen MP und MDP in den Abbildungen 9.13 und 9.14 zeigen.

In einer marktreifen Version der TIA, die für eine Testfallselektion und -priorisierung verwendet wird, sollte die Beachtung solcher Methoden trotzdem implementiert sein, da manche Projekte (wie zum Beispiel Achilles in dieser Studie) verstärkten Gebrauch davon machen.

Weiterhin sollte überlegt werden, ob und wie weitere Abhängigkeiten, die Bedingungen für das Ausführen der Tests einer Testklasse sind, registriert und beachtet werden können. Eine solche wurde beispielsweise im Studienobjekt Teamscale beobachtet: Hier wird für die UI-Tests ein Testserver gestartet, und ein einzelner UI-Test greift auf diesen vorher

gestarteten Server zu. Dies nachzuverfolgen ist in der derzeitigen Implementierung nicht möglich und voraussichtlich auch schwerlich umzusetzen sein, plant man eine Kompatibilität über viele Projektsetups hinweg zu erreichen.

10.4.3 RQ 3.3: Was sind die Auswirkungen einer unregelmäßigen Aktualisierung von Test-zu-Methoden-Beziehungen?

In der hier vorliegenden Studie konnte bei der Testdatenaufzeichnungen kein kritischer Ressourcen-Overhead (Zeit und Rechenleistung) beobachtet werden. Für Projekte, in denen aufgrund komplizierter Anwendung oder stark verlangsamter Ausführung die Aufzeichnungen nicht regelmäßig stattfinden, können die Erkenntnisse aus Forschungsfrage RQ 3.3 darauf hinweisen, dass die Anwendung der TIA selbst mit älteren Daten erfolgreich sein kann.

In keinem Fall ist die Time to Failure unter Verwendung der älteren Testausführungsinformation nennenswert angestiegen. In einigen Fällen ist sie sogar niedriger als bei Verwendung der letztmöglichen Coveragedaten. Allerdings lassen die fünf (von 15) nicht aufgedeckten Testfehlschläge vermuten, dass mit dem Alter der genutzten Testausführungsinformationen das Risiko steigt, Testfehlschläge nicht aufzudecken. Ein möglicher Grund hierfür sind Tests, die in der Zwischenzeit hinzugefügt wurden und die relevanten Codeänderungen abdecken. Eine andere Ursache kann das gewählte Studiensetup sein. Es wurden lediglich drei Codestände verwendet: der Stand, zu dem die älteste Testausführungsinformation erhoben wurde, der Stand des ursprünglichen letzten erfolgreichen Build vor dem fehlschlagenden Build und der Stand zu letztgenanntem Build. Teamscale kann auch bei der Änderung an einer Methode Coverageinformationen „mittracken“. Allerdings liegen zwischen dem ältesten verwendeten Codestand und dem ursprünglich vorhergehenden erfolgreichen Build einige Coderevisionen. Ohne diese Zwischenstände wird ein erfolgreiches Tracking möglicherweise verhindert.

Es zeigte sich kein Zusammenhang von der Zeitspanne zwischen genutzter Testausführungsinformation und Zeitpunkt des fehlschlagenden Builds mit dem Nichtausführen des fehlgeschlagenen Tests.

10.5 Beschränkung der Studie auf Regressionsfehler

Für die Daten aus Travis CI gilt, dass ein Buildfehlschlag aufgrund eines Tests nicht zwingend durch durch einen Regressionsfehler ausgelöst wurde. Fehlende Performanz von Testpriorisierungsverfahren in solchen Fällen lässt sich mit dem fehlenden Zusammenhang zu einer Codeänderung begründen. Diese Argumentation berücksichtigt allerdings nicht die Tatsache, dass Priorisierungsalgorithmen in der real stattfindenden Softwareentwicklung eingesetzt werden und dem Buildserver ermöglichen sollen, schnelles Feedback an den Entwickler zu senden.

Das hier angewendete Studiendesign unter Verwendung von Travis CI Buildfehlschlägen und Priorisierungsverfahren, die auf Code oder Ausführungsinformationen beruhen, ist also durchaus begründet.

Häufig sind auch flackernde Tests – also Tests, die bei mehrfacher Ausführung ohne Codeänderung zu unterschiedlichen Testresultaten führen – in Testsystemen problematisch. Eine generelle Aussage darüber zu treffen, wie ein Testselektion- oder Priorisierungsverfahren mit solchen umzugehen hat, ist schwierig und wenn überhaupt vermutlich nur für spezifische Projekte möglich. Die Ursachen für flackernde Tests sind unterschiedlich [Luo+14] und machen zum Beispiel bei Google mit 73 000 Fehlschlägen 4,56% der täglichen Testfehlschläge aus [Luo+14]. Eine Testinfrastruktur, die sowohl hardwareseitig

funktioniert, die Ausfälle von Drittsystemen kompensieren kann und in der sich deterministrische Testresultate zeigen, ist für effizientes Testen von großem Vorteil. Sollen flackernde Tests nicht etwa durch Mittel des Testsystems, zum Beispiel durch das Setzen von `rerunFailingTestsCount` (Maven Surefire Plugin) behandelt werden, sondern die Ursache der wechselnden Testresultate behoben werden, so ist fraglich, ob ein Testpriorisierungsverfahren solche Tests früh zur Ausführung vorschlagen soll.

10.6 Schwierigkeiten in der Datenerhebung und Auswertung

Obwohl in jüngerer Vergangenheit durch die Vorgängerarbeit [Rot18] bereits Erfahrungen hinsichtlich der nächträglichen Instrumentierung zur Testdatenaufzeichnung von GitHub-Travis-Projekten bestanden, wurde die Testdatenerhebung im Rahmen dieser Arbeit diffizil.

Kompatibilität Das neue Werkzeug zur Testdaten-Aufzeichnung erfordert (zum Beispiel durch die Unterstützung von maven-failsafe) eine leicht aufwändigere Konfiguration in der pom.xml der untersuchten Projekte. Dies führte in der großflächigen Datenerhebung zu Problemen, da die out-of-the-box Kompatibilität mit unterschiedlichen Projektkonfigurationen unterschiedlich war. Vor der Datenerhebung wurde die automatisch angepasste Konfiguration mit mehreren stichprobenartig gewählten Repository-Ständen erfolgreich getestet. Die Schwierigkeiten hinsichtlich der Kompatibilität (teils im selben, teils in anderen Projekten) hatten allerdings zur Folge, dass die Datenerhebung für einige zu untersuchende Builds nicht erfolgreich war und wiederholt werden musste.

Skalierbarkeit der Time to Failure Berechnung Wie in Kapitel 6 beschrieben, erfolgte die Berechnung auf einer groß dimensionierten Google Instanz. Dadurch konnte ein Geschwindigkeitsgewinn im Vergleich zu Recheninstanzen mit weniger Rechenleistung und Arbeitsspeicher verzeichnet werden. Unter anderem die Projekterzeugung bei Teamscale nahm einige Zeit in Anspruch. Dies mag eine Folge davon sein, dass das implementierte Programm zur Automatisierung zeitgleich mehr als 100 Projekte anlegte und daraufhin die Last auf dem Speichermedium sehr hoch war.

Speicherung der Benchmarkergebnisse Die Benchmarkergebnisse wurden, wie in der Vorgängerarbeit auch, in einer MySQL Datenbank gespeichert. Abweichend vom letztmals gewählten Verfahren wurden mehr Rohdaten in Form von zu JSON serialisierten Java-Objekten ablegt. So speicherte das entwickelte Programm

- die komplette geordnete Testausführungsliste für jedes Tripel bestehend aus
  - dem untersuchten Build,
  - dem eingesetzten Priorisierungsverfahren und
  - den zusätzlichen Parametern zur Konfiguration des Priorisierungsverfahrens.
- nicht nur die Zeit bis zum ersten Fehlschlag pro Algorithmus, sondern pro Build die Zeiten bis zu jedem Testfehlschlag, der bei diesem Build laut Travis auftrat.

Diese Methode wurde gewählt, um die Rohdaten zur Verfügung zu haben, falls sie zu einem späteren Zeitpunkt erneut gebraucht werden sollten (Debugging, Wiederholung des Benchmarks anhand der gespeicherten Liste anstatt diese nochmalig von Teamscale berechnen zu lassen, Erhebung von Meta-Daten).

Die Vorteile dieses Speicherverfahrens überwogen im Rahmen der Studie dessen Nachteile nicht. Die Speicherung als serialisierte Objekte machten die Auswertung der auf

dem Datenbankserver gespeicherten Resultate aufwändiger, da ein zusätzlicher Verarbeitungsschritt zur Deserialisierung eingebaut werden musste. Ebenfalls wäre durch die Speicherung von ready-to-use Resultaten ein Überblick durch die Sichtung einzelner Tabellen schneller möglich gewesen.

Datenverluste durch Timeouts Wie in Abschnitt 7.3.4 beschrieben, stellte das Benchmark-Tooling zur Berechnung der Time to Failures Anfragen an den auf derselben Instanz laufenden Teamscale-Server. Der Zeitwert, nachdem die Anfrage wegen eines Timeouts abgebrochen werden sollte, wurde auf 15 Minuten gesetzt. Vereinzelt traten diese Timeouts auf, was zur Folge hatte, dass die Daten, die als Resultat auf die Anfrage folgten, nicht im Benchmark verwendet wurden. Eine so lange Priorisierungszeit wurde auch für große Projekte nicht erwartet. A posteriori mag dies allerdings für ungünstige Kombinationen zeitintensiver Berechnungen (zum Beispiel in großen Projekten die Berechnung der String-Distanzmatrix über alle vorhandenen Methoden) und Anfragen an die Datenbank (um die Menge an hinzugefügten und gelöschten Tests zu errechnen; siehe Abschnitt 4.3.5) doch vorkommen. Es wurde beobachtet, dass die Last auf dem Teamscale-Server auch nach Abbruch der Service-Aufrufe hoch bleibt. Die Vermutung liegt nahe, dass die Berechnung also trotz Abbruch fortgeführt wird, danach allerdings verloren geht und gleichzeitig durch die hohe Last zur Verursachung weiterer Timeouts beiträgt.

Vielfalt der implementatorischen Anpassungen und benutzten Tools Dritter Weiterhin umfasste diese Arbeit nicht lediglich die empirische Evaluation von Priorisierungsalgorithmen, sondern auch die Integration von Algorithmen in das darunter liegende System und die Umarbeitung desselben. Durch den vermehrten Einsatz von Implementationen Dritter, die sich zumeist in einem Beta-Entwicklungsstadium befanden, mussten viele Eigenheiten beachtet und einige Programmabschnitte angepasst werden.

10.7 Einschränkungen der Validität

Konzept und Implementation dieser Arbeit wurden mit großer Sorgfalt entwickelt. Trotz allem unterliegt die Arbeit einigen Einschränkungen in der Validität, die in den folgenden Paragraphen dargelegt werden.

Implementation der Priorisierungsalgorithmen Die Implementation der Priorisierung im Analysewerkzeug Teamscale war zum Teil bereits vorhanden. Im Rahmen der Studie wurden die geschilderten Neuerungen und Erweiterungen programmiert. Trotz sorgfältiger Tests ist es möglich, dass Fehler im System unentdeckt blieben. Soweit vorhanden, beeinflussten diese die berechneten Testausführungslisten und damit die Time to Failure und die Resultate des Benchmarks.

Implementation des Benchmarkverfahrens Das Benchmarkverfahren für die Builds mit den real aufgetretenen Testfehlern wurde im Rahmen dieser Arbeit erstellt. Der Mutations-Benchmark wurde von Niedermayr übernommen, wie beschrieben angepasst und die Daten aus den ausgegeben Dateien durch ein eigens implementiertes Tooling extrahiert. Sowohl in den eigenen als auch in den von Dritten programmierten Systemen können Fehler enthalten sein, die unentdeckt blieben. Das gleiche gilt für die entwickelten Auswertungsmechanismen.

Coverageerhebung Die Coverageerhebung im Benchmark mit den real auftretenden Testfehlern erfolgte zum großen Teil automatisiert. Es wurde versucht, die automatische Konfigurationsanpassung möglichst kompatibel mit vielen Projekten zu gestalten, und es wurden nur die Builds im Benchmark verwendet, für die Coverage erhoben werden konnte. Fehlerhaft oder unvollständig erhobene Coverage kann trotzdem nicht ausgeschlossen werden. Stichprobenweise wurde kontrolliert, wie sehr sich Coveragedaten innerhalb eines Projektes in der Größe unterschieden und ob die testspezische Coverage in Teamscale entsprechend importiert und dargestellt wurde. Die hohe Anzahl an Builds, für die auch unter Verwendung des Selektionsalgorithmus der fehlschlagende Test ausgeführt wurde, spricht dafür, dass die Coverageerhebung im Generellen gut funktioniert hat. Ohne Coverage kann von einer geänderten Methode nicht auf einen auszuführenden Test geschlossen werden, und für eine größere Anzahl an Builds wären Testfehlschläge unentdeckt geblieben.

Datenqualität der Mutationsmatrizen und der Stack-Distanz-Werte Die Erzeugung von Mutationsmatrizen ist ein aufwändiger Prozess und in unterschiedlichen Projekten verschieden gut möglich. Die Vollständigkeit und Güte der in der Priorisierung verwendeten externen Daten konnte im Rahmen der vorliegenden Arbeit nicht überprüft werden. Die von Niedermayr erhaltenen Daten (Mutationsmatrizen für einige Projekte und Stack-Distanz-Werte) finden auch in weiteren Forschungsarbeiten Anwendung.

Dauer einzelner Tests Die stärksten absoluten Einsparungen können in der TIA nur dann erreicht werden, wenn die auszuführenden Tests eine große Ausführungsdauer haben. In der hier vorliegenden Studie wurden vor allem Projekte mit nicht lang andauernden Tests untersucht. Ergebnisse aus dieser Studie sind nicht generell auf Projekte mit langen Tests übertragbar.

Möglichkeit der Ausführung der Tests in der vorgeschlagenen Reihenfolge Die Realitätsoptionen wurden in Zusammenarbeit mit Mitarbeitern der CQSE GmbH entworfen. Obwohl darauf geachtet wurde, in der Studie eine plausible Realitätskonfiguration zu verwenden, kann es sein, dass Tests aufgrund von technischen Beschränkungen nicht in der vorgeschlagenen Reihenfolge ausgeführt werden können. Dies hat einen Einfluss auf die Time to Failure Werte und deren Auswertung in der Evaluation der einzelnen Priorisierungsalgorithmen.

11 Fazit

In dieser Arbeit wurden Testselektions- und Priorisierungsverfahren anhand zweier verschiedener Arten von Benchmark-Systemen evaluiert sowie zusätzliche Fragestellungen hinsichtlich technischer oder prozeduraler Aspekte beantwortet.

Die Performanz von Priorisierungsverfahren wurde einerseits anhand von 24 GitHub-Travis-Projekten untersucht, wobei Builds verwendet wurden, die auf dem CI-Server einen tatsächlichen Testfehlschlag erfuhren. Andererseits wurde ein Benchmark auf zwölf Projekten ausgeführt, der mit Daten aus Mutationsanalysen arbeitet. Zusätzlich wurden beide Benchmark-Systeme auch auf das Studienobjekt Teamscale angewandt. Neben Teamscale enthielt die Schnittmenge der in beiden Benchmarks untersuchen Projekte 8 GitHub-Travis-Projekte.

Insgesamt unterstützen die erhobenen Daten die Annahme, dass die Verwendung von Selektions- oder Priorisierungsverfahren beziehungsweise Kombinationen solcher Verfahren von Vorteil ist, sollen mögliche Testfehler früh entdeckt werden. Dies kann Entwicklern die Arbeit erleichtern und die Projektqualität konstant halten oder steigern.

Zudem wurden anhand von drei Projekten die Ergebnisse aus dem Benchmark mit real auftretenden Testfehlschlägen mit Ergebnissen aus dem Mutations-Benchmark hinsichtlich des in der Forschungsgruppe vorgeschlagenen hybriden Selektions- und Priorisierungsverfahrens EACPTS verglichen. In zwei von drei Projekten zeigten sich unter Anwendung dieses Verfahrens die niedrigsten Time to Failure-Werte.

Die Ergebnisse sprechen zudem dafür, dass eine kontinuierliche Erhebung von Testausführungsinformationen zur erfolgreichen Anwendung der EACPTS nicht zwingend notwendig ist, möglicherweise allerdings ein erhöhtes Risiko mit sich bringt, eigentlich fehlschlagende Tests nicht auszuführen.

Nach wie vor sollte die Forschung auf diesem spannenden Gebiet vorangetrieben werden. Einstiegspunkte und Ideen sind auch im abschließenden nächsten Kapitel zu finden.

12 Ausblick

Die vorliegende Studie schließt zusammen mit der Vorgängerarbeit [Rot18] eine Lücke in der Forschung. Priorisierungsalgorithmen auf der Basis von historischen Buildfehlschlägen zu evaluieren und diese Ergebnisse mit Mutationsanalysen auf gleichen Projekten (das gilt in dieser Studie für eine Teilmenge der Forschungsobjekte) zu vergleichen, ist neu.

Das Potential der Nutzung von Testpriorisierung hat sich an einigen Stellen gezeigt. Verschiedenerlei Möglichkeiten, weiter in diese Richtung zu forschen, stehen offen. Einige von ihnen sowie Möglichkeiten, in folgenden Durchführungen der Studie das Design anzupassen, seien nun noch dargelegt.

Weitere Eingabedaten zur Verfeinerung der Priorisierungslogik In Abschnitt 7.3.2 wurden vier – in dieser Studie hinzugezogene – Eingabedaten vorgestellt. Für weitere Forschung wird vorgeschlagen, weitere Aspekte bei der Priorisierung zu berücksichtigen.

Dies kann zum einen eine Information darüber sein, wie häufig ein Test fehlschlägt. Neigt ein Test zu häufigen Fehlschlägen, kann es, möchte man möglichst schnell den ersten Testfehlschlag erreichen, sinnvoll sein, ihn im Testprozess frühzeitig auszuwählen und auszuführen. Im Rahmen der vorliegenden Arbeit wurde bereits ein Werkzeug implementiert, das Testfehlschläge, die bei Travis Builds aufgetreten sind, auf ein für Teamscale lesbares Format konvertieren kann.

Störend bei der Berücksichtigung von Testfehlschlägen sind allerdings flackernde Tests, da sie möglicherweise hinsichtlich des Code-Deltas unberechtigt selektiert oder früh ausgeführt werden. Auch aus diesem Grund ist es ratsam, die Test-Suite eines Projekts möglichst frei von flackernden Tests zu halten und bei deren Auftreten die Ursache frühzeitig auszumachen und zu beheben.

Im Kontext dieser Arbeit beschränkt sich die TIA auf Änderungen im Java-Quelltext. Das bedeutet insbesondere, dass Änderungen an Konfigurations- oder Testdaten (zum Beispiel an Ergebnisdateien oder einem Archiv, das m Test verwendet wird) nicht zur Priorisierung verwendet werden können. Durch eine entsprechende Instrumentierung des System unter Tests (zum Beispiel durch Verwendung von `strace`) kann für Testfälle die Abhängigkeit von anderen im Repository befindlichen Dateien extrahiert werden.

Außerdem ist die Integration eines Algorithmus denkbar, der die Testselektion und -priorisierung mittels Machine Learning verbessern soll. Innerhalb eines großen Softwareunternehmens wurde bereits daran geforscht, und es könnte sich (hinsichtlich der gesamten Testinfrastruktur) eine Kostenersparnis von etwa 50% ergeben [Mac+18].

Weitere Untersuchung für parametrisierte Tests Nicht untersucht wurde im Zusammenhang mit dieser Arbeit die Aufzeichnung von Zuliefermethoden von parametrisierten Tests. Es muss geklärt werden, ob diese ähnlich wie @Before-Methoden für unparametrisierte Tests bereits den richtigen Testfällen zugeordnet werden oder ob eine Nachjustierung an der Aufzeichnung und in der TIA-Implementierung erforderlich ist.

Machbarkeitsstudie von verschiedenen Realitätsoptionen Die im Laufe dieser Arbeit vorgestellten und verwendeten Realitätsoptionen beschreiben einen Aspekt des Leistungsumfang des Testsystems. Im Rahmen der vorliegenden Studie wurden aus Forschungsgründen neben dem derzeitigen Standard, der vom Maven Surefire Plugin unterstützt wird, unterschiedliche Varianten (zwei Lockerungen in den Bedingungen) getestet. Wie in Kapitel 9 vorgestellt, zeigen sich durch die unterschiedliche Anwendung auch stark variierende Times to Failure.

In einer zukünftigen Studie gilt es zu untersuchen, wie bisher nicht verwirklichte Realitätsoptionen, die einen Vorteil durch geringere Testzeiten bringen, implementiert werden können. Dabei stellt sich die Frage, wo feste Grenzen liegen, die nicht mehr durch technische Verbesserungen, sondern durch logische Bedingungen gezogen werden. So können Abhängigkeiten zwischen Tests nicht durch reine Verbesserung des testausführenden Systems überwunden werden. Das Hin- und Herwechseln zwischen Testfällen verschiedener Testklassen ist jedoch durch länger im Speicher gehaltene Systemzustände und Daten denkbar.

Längerfristig gültige Ausführungsreihenfolgen Die Ergebnisse der Forschungsfrage 3.3 legen nahe, dass die Coverageaufzeichnung nicht bei jeder Testausführung notwendig ist, um gute Ergebnisse in der Priorisierung zu erreichen.

In zukünftiger Forschung sollte geklärt werden, ob sich dies noch weiter „vereinfachen“ lässt, etwa in Form einer einmalig erstellten Testausführungsreihenfolge, die eine längerfristige Gültigkeit besitzt. Hierfür könnten (vermutlich in Abhängigkeit von der Entwicklungsaktivität) beispielsweise Größenordnungen von mehreren Wochen bis hin zu einem Jahr untersucht werden. Dies ist insbesondere für Projekte wichtig, in denen die Coverageerhebung sehr aufwändig und kostspielig ist.

Einfluss der Testgestaltung auf die Wirksamkeit von Priorisierungsverfahren Daten aus der Vorgängerarbeit stützen die Vermutung, dass Tests, die einen kleineren Anteil vom System abdecken, besser durch TIA priorisiert werden können als Tests, die einen sehr großen Teil des Systems covern. Für die weitere TIA Forschung ist es relevant zu wissen, welche Einflüsse das prinzipielle Testdesign (Architektur, Größen) und die Testinfrastruktur (Testsystem) haben.

Entwicklungsbegleitende Evaluation und Optimierung anhand Projektspezifika Der an der Forschungsgruppe vorgestellte Selektions- und Prioriserungsalgorithmus wurde in dieser Studie und in weiteren studentischen Arbeiten (vgl. [Dre17; Ein18; Rot18]) mit Mutationsanalysen und mit einem Einsatz im Bereich eingebetteter Systeme auf die Probe gestellt.

Die Einführung des Algorithmus (und dessen Varianten/Verfeinerungen durch Nutzung von Stack-Distanz- oder Mutations-Daten) und die andauernde Nutzung in einem (Industrie)Projekt mit entsprechend hoher Entwicklungsaktivität in einer Langzeitstudie zu evaluieren, wäre begrüßenswert.

Dies ermöglichte die fokussierte Betrachtung der Performanz innerhalb einer Umgebung. Dies ist mitunter deshalb sinnvoll, da der Algorithmus in dieser und in der Vorgängerstudie in unterschiedlichen Projekten verschiendene relative Time to Failure-Werte zeigte.

Möglichkeiten der Feinjustierung, zum Beispiel die Berücksichtigung von vergangenen Testfehlschlägen oder zusätzlichen manuellen Tests, könnten durch projektnahe Forschung besser betrachtet werden. Eine gefürchtete Hemmschwelle mag der initiale Ein-

richtungaufwand eines Industrieprojekts sein. Allerdings wird dieser durch voraussichtlich profitable Verkürzungen der Test- und damit auch Buildzeiten kompensiert.

Erweiterung um zusätzliche Priorisierungsalgorithmen Die Vorgängerstudie zur Effizienz von Testpriorisierungsalgorithmen legt nahe, dass Priorisierungsstrategien gleichermaßen gut in unterschiedlichen Projekten arbeiten.

Die verhältnismäßig geringe Anzahl an verfügbaren Builds für einige Projekte war Grund dafür, dass in der hier vorliegenden Studie keine Priorisierungsstrategien implementiert wurden, die maschinelles Lernen als algorithmischen Ansatz verwenden. Erlaubt es die Datenlage oder evaluiert man in einem projektspezifischen Kontext, dann sollten auch Verfahren, die nach der vorgenannten Art und Weise arbeiten, untersucht werden.

Bedeutung von TIA für manuelle Tests Obwohl die Testautomatisierung ein wichtiges Instrument ist, um regelmäßig und effizient zu testen, werden in einigen Softwareprojekten der Industrie Tests manuell ausgeführt. Dabei folgen Tester dem Ablauf, der in definierten Testfällen festgehalten ist. Manuelle Tests sind durch die Ausführung per Hand in der Regel jedoch teurer und langsamer.

Aus diesem Grund ist die Priorisierung und Selektierung vor der Testausführung hier besonders wichtig. Für künftige Arbeiten wäre es spannend zu untersuchen, wie die in der Forschungsgruppe entwickleten Methoden und deren vorgestellte Variationen die Effizienz im manuellen Testprozess beeinflussen.

Reproduktion mit anderen Studienobjekten Die Ergebnisse der hier dargelegten Studie wurden auf Basis der in Kapitel 7 beschriebenen Projekte und Builds (für den Benchmark mit realen Testfehlern) und unter Verwendung von Builds mit eingestreuten Mutationen erzielt. Wie in Abschnitt 10.7 erwähnt, lassen diese Daten eine Verallgemeinerung im Generellen nicht zu.

Da je nach verwendeten Realitätsoptionen die Ergebnisse variieren und unterschiedlich deutliche Effekte zeigen, sollte eine Reproduktion der Evaluation von verschiedenen Testpriorisierungsverfahren durchgeführt werden. Zur Gewinnung der Daten können dabei natürlich auch Quellen jenseits von GitHub und Travis CI herangezogen werden. Das vorgestellte Benchmarksystem bietet hierfür eine solide Grundlage.

Anhang

A Beispiel: Kein Code-Delta – Ansicht in GitHub und Teamscale-Aktivitätsperspektive

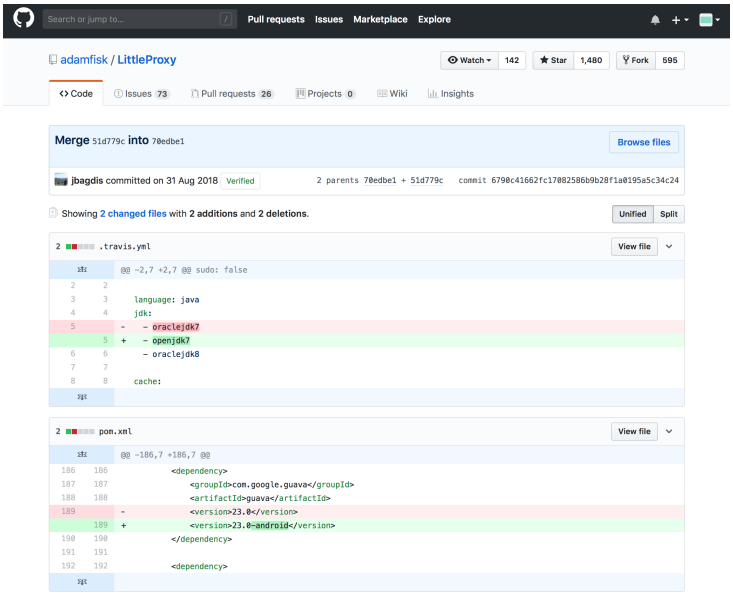


Abbildung A.1: Änderung abseits von Quellcode im Repository, die einen roten Build zur Folge hatte.

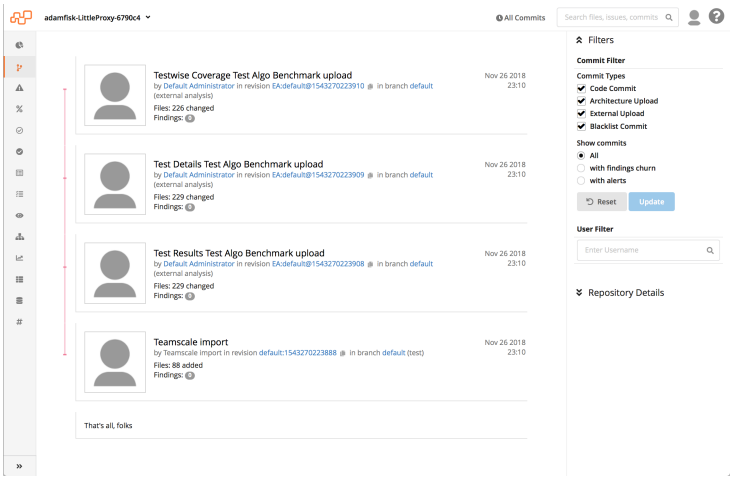


Abbildung A.2: In der Teamscale-Aktivitätsansicht ist kein zweiter Codecommit zu sehen. Der betreffende Build wurde von der Analyse ausgeschlossen.

B Vergrößerter Boxplot für EACPTS-Variationen im Projekt Teamscale

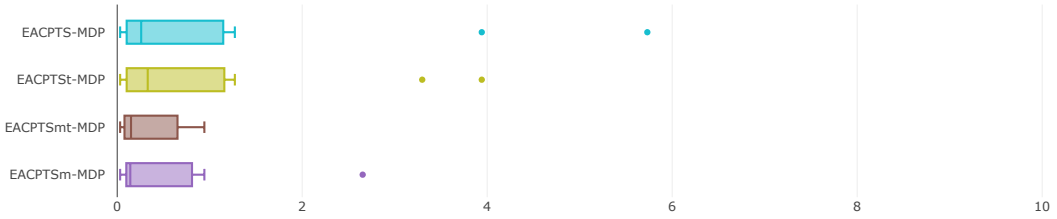


Abbildung B.1: Vergrößerter Ausschnitt der Boxplots für die Leistung der EACPTS-Variationen im mutationsbasierten Benchmark für das Projekt *teamscale* mit 5 zusätzlich als geändert markierten Methoden.

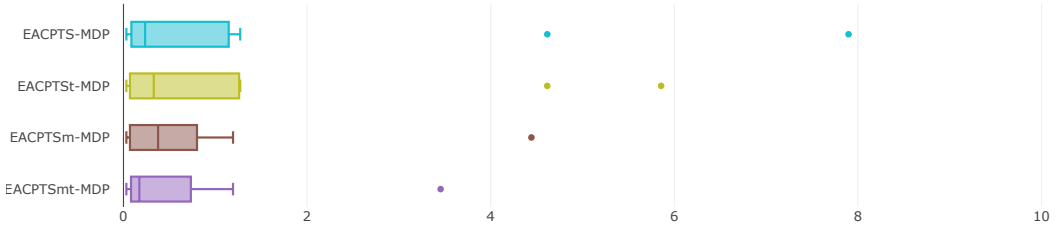


Abbildung B.2: Vergrößerter Ausschnitt der Boxplots für die Leistung der EACPTS-Variationen im mutationsbasierten Benchmark für das Projekt *teamscale* mit 10 zusätzlich als geändert markierten Methoden.

Literatur

[BGZ17a] Beller, Moritz – Gousios, Georgios – Zaidman, Andy. "Oops, my tests broke the build: An explorative analysis of Travis CI with GitHub". In: *Mining Software Repositories (MSR), 2017 IEEE/ACM 14th International Conference on*. IEEE. 2017, S. 356–367.

[BGZ17b] Beller, Moritz – Gousios, Georgios – Zaidman, Andy. "TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration". In: *Proceedings of the 14th working conference on mining software repositories*. 2017.

[Boo91] Booch, Grady. *Object Oriented Design with Applications*. Redwood City, CA, USA: Benjamin-Cummings Publishing Co., Inc., 1991. ISBN: 0-8053-0091-0.

[CRV94] Chen, Yih-Farn – Rosenblum, David S – Vo, Kiem-Phong. "TestTube: A system for selective regression testing". In: *Proceedings of the 16th international conference on Software engineering*. IEEE Computer Society Press. 1994, S. 211–220.

[DMB12] Dösinger, Stefan – Mordinyi, Richard – Biffl, Stefan. "Communicating continuous integration servers for increasing effectiveness of automated testing". In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM. 2012, S. 374–377.

[Dre17] Dreier, Florian. "Obtaining Coverage per Test Case". Master's Thesis. Technische Universität München, 2017.

[Ein18] Einwang, Stefanie. "Empirische Untersuchung der Priorisierung von automatisierten Tests für eingebettete Systeme auf Basis kürzlich durchgeführter Code-Änderungen". Master's Thesis. Technische Universität München, 2018.

[Elb+04] Elbaum, Sebastian – Rothermel, Gregg – Kanduri, Satya – Malishevsky, Alexey G. "Selecting a cost-effective test case prioritization technique". In: *Software Quality Journal* 12.3 (2004), S. 185–210.

[FF06] Fowler, Martin – Foemmel, Matthew. "Continuous integration". In: *ThoughtWorks) http://www. thoughtworks. com/Continuous Integration. pdf* 122 (2006), S. 14.

[FRC81] Fischer, Kurt – Raji, Farzad – Chruscicki, Andrew. "A methodology for retesting modified software". In: *Proceedings of the National Telecommunications Conference B-6-3*. 1981, S. 1–6.

[Gra+01] Graves, Todd L – Harrold, Mary Jean – Kim, Jung-Min – Porter, Adam – Rothermel, Gregg. "An empirical study of regression test selection techniques". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 10.2 (2001), S. 184–208.

[Hsi+97] Hsia, Pei – Li, Xiaolin – Chenho Kung, David – Hsu, Chih-Tung – Li, Liang – Toyoshima, Yasufumi – Chen, Cris. "A technique for the selective revalidation of OO software". In: *Journal of Software Maintenance: Research and Practice* 9.4 (1997), S. 217–233.

[Jür+11] Jürgens, Elmar – Hummel, Benjamin – Deissenboeck, Florian – Feilkas, Martin – Schlögel, Christian – Wübbeke, Andreas. "Regression test selection of manual system tests in practice". In: *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*. IEEE. 2011, S. 309–312.

[Lu+16] Lu, Yafeng – Lou, Yiling – Cheng, Shiyang – Zhang, Lingming – Hao, Dan – Zhou, Yangfan – Zhang, Lu. "How does regression test prioritization perform in real-world software evolution?" In: *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE. 2016, S. 535–546.

[Luo+14] Luo, Qingzhou – Hariri, Farah – Eloussi, Lamyaa – Marinov, Darko. "An Empirical Analysis of Flaky Tests". In: *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. Hong Kong, China: ACM, 2014, S. 643–653. ISBN: 978-1-4503-3056-5.

[Mac+18] Machalica, Mateusz – Samylkin, Alex – Porth, Meredith – Chandra, Satish. "Predictive Test Selection". In: *arXiv:1810.05286* (2018).

[NJW16] Niedermayr, Rainer – Jürgens, Elmar – Wagner, Stefan. "Will My Tests Tell Me If I Break This Code?" In: *Proceedings of the International Workshop on Continuous Software Evolution and Delivery (CSED'16)*. ACM. 2016.

[NRW18a] Niedermayr, Rainer – Röhm, Tobias – Wagner, Stefan. "Poster: Identification of Methods with Low Fault Risk". In: *Proceedings of the 40th International Conference on Software Engineering Companion (ICSE'18)*. ACM. 2018.

[NRW18b] Niedermayr, Rainer – Röhm, Tobias – Wagner, Stefan. "Too Trivial To Test? An Inverse View on Defect Prediction to Identify Methods with Low Fault Risk". In: *CoRR* abs/1811.00820 (2018).

[NW] Niedermayr, Rainer – Wagner, Stefan. "Is the Stack Distance Between Test Case and Method Correlated With Test Effectiveness?"

[Ras04] Rasmusson, Jonathan. "Long build trouble shooting guide". In: *Conference on Extreme Programming and Agile Methods*. Springer. 2004, S. 13–21.

[RH96] Rothermel, Gregg – Harrold, Mary Jean. "Analyzing regression test selection techniques". In: *IEEE Transactions on software engineering* 22.8 (1996), S. 529–551.

[Rot+01] Rothermel, Gregg – Untch, Roland H. – Chu, Chengyun – Harrold, Mary Jean. "Prioritizing test cases for regression testing". In: *IEEE Transactions on software engineering* 27.10 (2001), S. 929–948.

[Rot+17] Rott, Jakob – Niedermayr, Rainer – Jürgens, Elmar – Pagano, Dennis. "Ticket coverage: putting test coverage into context". In: *2017 IEEE/ACM 8th Workshop on Emerging Trends in Software Metrics (WETSoM)*. IEEE. 2017.

[Rot+99] Rothermel, Gregg – Untch, Roland H – Chu, Chengyun – Harrold, Mary Jean. "Test case prioritization: An empirical study". In: *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*. IEEE. 1999, S. 179–188.

[Rot18] Rott, Jakob. "Building a benchmark to compare strategies of test prioritization". Guided Research. Technische Universität München, 2018.

[SB14] Ståhl, Daniel – Bosch, Jan. "Modeling continuous integration practice differences in industry software development". In: *Journal of Systems and Software* 87 (2014), S. 48–59.

[She95] Sherlund, Basil Anton. "Logical Modification Oriented Regression Testing". In: *Defect prevention: presentations of the 12th International Conference and Exposition on Testing Computer Software*. 1995, S. 287–303.

[Tuf+17] Tufano, Michele – Palomba, Fabio – Bavota, Gabriele – Di Penta, Massimiliano – Oliveto, Rocco – De Lucia, Andrea – Poshyvanyk, Denys. "There and back again: Can you compile that snapshot?" In: *Journal of Software: Evolution and Process* 29.4 (2017), e1838.

[YH12] Yoo, Shin – Harman, Mark. "Regression testing minimization, selection and prioritization: a survey". In: *Software Testing, Verification and Reliability* 22.2 (2012), S. 67–120.