RTPTorrent: An Open-source Dataset for Evaluating Regression Test Prioritization

Toni Mattis
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
toni.mattis@hpi.uni-potsdam.de

Patrick Rein
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
patrick.rein@hpi.uni-potsdam.de

Falco Dürsch
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany

Robert Hirschfeld
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
hirschfeld@hpi.uni-potsdam.de

ABSTRACT

The software engineering practice of automated testing helps programmers find defects earlier during development. With growing software projects and longer-running test suites, frequency and immediacy of feedback decline, thereby making defects harder to repair. Regression test prioritization (RTP) is concerned with running relevant tests earlier to lower the costs of defect localization and to improve feedback.

Finding representative data to evaluate RTP techniques is non-trivial, as most software is published without failing tests. In this work, we systematically survey a wide range of RTP literature regarding whether their dataset uses real or synthetic defects or tests, whether they are publicly available, and whether datasets are reused. We observed that some datasets are reused, however, many projects study only few projects and these rarely resemble real-world development activity.

In light of these threats to ecological validity, we describe the construction and characteristics of a new dataset, named RTPTorrent, based on 20 open-source Java programs.

Our dataset allows researchers to evaluate prioritization heuristics based on version control meta-data, source code, and test results from fine-grained, automated builds over 9 years of development history. We provide reproducible baselines for initial comparisons and make all data publicly available.

We see this as a step towards better reproducibility, ecological validity, and long-term availability of studied software in the field of test prioritization.

MSR '20, October 5–6, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7517-7/20/05…$15.00
https://doi.org/10.1145/3379597.3387458

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **General and reference** → *Evaluation*; • **Information systems** → Data mining.

KEYWORDS

Regression Test Prioritization, TravisCI, GitHub, Java, Dataset

ACM Reference Format:
Toni Mattis, Patrick Rein, Falco Dürsch, and Robert Hirschfeld. 2020. RTPTorrent: An Open-source Dataset for Evaluating Regression Test Prioritization. In *17th Intern ational Conference on Mining Software Repositories (MSR '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3379597.3387458

1 MOTIVATION

Automated test suites for programs tend to grow continuously as software evolves, accumulating not only new requirements, but also an ever-growing amount of previously reproduced defects to prevent their re-introduction. As automated testing becomes costlier, developers shy away from running time-consuming test suites in their development environments and Continuous Integration (CI) infrastructure suffers longer *build cycles*. The benefits of obtaining rapid feedback whether a change introduced a regression wane and defects get harder to detect and repair [43].

The field of Regression Test Prioritization (RTP) addresses the challenges of delayed feedback and computational costs caused by long-running test cycles [69, 92]. RTP generally uses *heuristics* to predict the fault-detection capability of individual tests or whole test schedules.

Manifold RTP techniques have been proposed in recent years [69]. Devising general RTP techniques which are effective in a variety of settings is challenging, as there are several trade-offs involved, for example between the effort required for gathering data to inform the prioritization, and the actual gains of the prioritization. The requirement of being applicable in a wide range of settings, and the fact that RTP generally uses heuristics, renders the evaluation of RTP techniques difficult [38, 72].

State of the Art in Regression Test Prioritization. Most evaluation studies on the effects of a particular RTP techniques require a

dataset that includes the source code of a program, a pool of test cases, and a set of faults. In concordance with the goal of the evaluation, these datasets should either have well-known properties to allow for controlled experimentation, or resemble real-world projects to assess the utility of techniques under real-world conditions. Further, datasets should be archived and openly available to make the results of different studies comparable [15].

In order to assess the current state of the art in RTP evaluation studies, we conducted a literature survey, which we describe in section 2. Our survey showed considerable reuse of datasets in the community. At the same time, it showed that most datasets contain a limited number of projects and many of these projects can not be considered to resemble real-world projects. While this might be desirable for controlled experiments in order to assess the theoretical performance of RTP techniques, it does at the same time impede the ecological validity of the results.

An Open Dataset. To complement existing datasets designed for controlled environments or synthetic test runs, we introduce the RTPTorrent dataset consisting of 20 open-source Java projects from *GitHub* with more than 100 000 real-world build jobs from the *TravisCI* continuous integration infrastructure[1]. Compared to all Java projects currently hosted on GitHub, these projects span a wide range with regard to size, number of contributors, and maturity. In section 3, we describe project selection, properties, qualitative characteristics, and our acquisition process.

Non-trivial Baseline. We run a micro-study implementing an effective prioritization heuristic based on test failure history to demonstrate the use of our dataset. We present its results in section 4 and include them in the dataset to provide a non-trivial baseline that complements commonly used random and default test-runner-determined baselines.

Contributions. In summary, our contributions comprise:

- A systematic review of datasets used throughout RTP literature from the perspective of re-using their study subjects and reproducing their results as baselines for future research.
- A novel Java-based open-source dataset addressing the concerns emerging from the literature study.
- A quantitative description of selected projects, including their representativeness with respect to all Java projects on GitHub.
- A prioritization micro-study that can serve as non-trivial baseline.

2 STATE OF THE ART DATASETS FOR TEST PRIORITIZATION

Datasets are important for the evaluation of RTP techniques, as the properties of the dataset might influence the impact on the performance of a RTP technique in practice. At the same time, the advancement of the field depends on comparable evaluations, which in turn requires that studies are conducted using common datasets. In order to assess the state of the art in datasets for RTP evaluation, we characterized the internal qualities of the used datasets, as well as the degree to which datasets are shared and archived. Therefore, we conducted a literature survey with regard to the following three research questions:

(1) What kind of projects, test cases, and faults are used to evaluate RTP techniques?
(2) Which datasets are shared amongst evaluation studies of RTP techniques?
(3) To which degree are the datasets available?

2.1 Methodology

In general, we followed the SALSA (Search, AppraisaL, Synthesis, Analysis) process for conducting the literature survey [34]. In the following, we describe the individual steps of the process and the respective intermediate sizes of the set of candidate publications.

Search. As the goal of the survey was to investigate datasets used for current research on RTP, we strove for a complete survey. As an initial sample we used the literature cited in a recent survey on RTP [69].

In order to get a complete picture of the current situation, we also retrieved all publications of selected publication venues since 2009, assuming that any regularly used datasets will also be used in recent publications. We selected the publication venues to retrieve the additional publications by choosing prominent software engineering publication venues and adding venues which published prominent papers in the field of RTP. Overall, we added publications from: Conference on Mining Software Repositories (MSR), Conference on the Foundations of Software Engineering (FSE), International Conference on Software Engineering (ICSE), International Symposium on Software Testing and Analysis (ISSTA), Symposium On Applied Computing (SAC), Tests and Proofs Conference (TAP), Journal of Systems and Software (JSS), International Conference on Software Security and Reliability / Secure Software Integration and Reliability Improvement (SSIRI/SERE).

We retrieved the bibliographic information of the publications of these venues through dblp[2]. We first fetched the full list of each publication venue via the corresponding overview page (for example dblp.uni-trier.de/db/conf/msr/msr2019.html), while accounting for special editions of the venue. We then parsed each page, extracted the URLs of the corresponding Bibtex resources, and downloaded the bibliographic information.

Combined with the literature from the survey [69], we retrieved a set of *10 308 publications*.

Appraisal. We were interested in publications specifically working on RTP. Thus, in an initial step, we reduced the number of publications by only selecting papers which included "test", "prioritiz", or "prioritis" in title or keywords. This reduced the candidate set to 1 484 publications (of which 1290 stem from our search, 205 from the 2019 survey, and 11 appeared in both sets).

In a second step, we reviewed each remaining candidate publication on whether it refers to general regression test prioritization. In particular, we rejected works on test suite reduction, test selection, or test case generation. Publications which described any of these approaches in combination with RTP were accepted. Further, as we were interested in datasets which are used for assessing general-purpose RTP approaches, we also excluded approaches targeting

[1] The dataset is available at doi.org/10.5281/zenodo.3712290.

[2] https://dblp.uni-trier.de

specific application domains, such as web service composition or mobile applications, or special types of tests, such as manual tests or feature interaction tests.

Finally, we removed all papers which did not describe a study using a dataset of projects including source code. The resulting set of publications considered in the survey includes *117 publications*.

Synthesis. To characterize the publications, we used thematic analysis [8]. When characterizing datasets, we surveyed each study reported in a publication separately. For research questions 1 and 3, we coded the publications using a pre-defined coding scheme (applying a theoretical thematic analysis [8]). We describe the individual dimensions and codes in section 2.2. In detail, the nature of programs, variants/versions, and faults are a recurring theme that tends to provoke discussions about validity and generality of RTP studies [19]. Moreover, the broad availability of fine-grained program versions (GHTorrent) and real build logs (TravisTorrent) calls the argument that manual or synthetic/generated data be easier to obtain into question, which prompted us to focus on these properties. We extended this to the nature of the tests used in studies, as these are also relevant for history-based approaches. The number of projects per dataset is a general aspect relevant to empirical studies. Finally, the degree of availability of datasets originates from our own difficulties of replicating previous work.

Our initial choice of codes was subject to little subjectivity, as authors state what data they use for evaluation (for example, "...variants of the program where faults have been seeded manually" [66]). The data was coded by one coder. Whenever a code could not be assigned unambiguously, we discussed the ambiguity with co-authors with the goal to refine the coding scheme, then re-coded the previous papers using the updated rules.

For research question 2, we first collected all existing datasets mentioned in publications. Then we derived individual names for the datasets, and finally coded the publications by assigning these names (thereby applying inductive thematic analysis [8]). In case a study used projects from several datasets, we coded the individual numbers of projects from each dataset.

Analysis. As we did not approach the literature survey with a specific hypothesis, we did not employ a fixed-setup analysis. Instead, we observed general patterns in the data (for the discussion of the results see section 2.3).

2.2 Classification of Datasets

Throughout the dimensions, we assume that a dataset consists of a number of *software projects*. These might come in different *variants* such as releases, commits, or generated mutants. Further, for each project there can be a number of *tests*.

In the following, we describe the single dimensions, the codes (For each dimension and code, we also provide the column name and symbol used in table 1.)

Research Question 1. To answer the first research question, we used the following dimensions with the described codes to characterize datasets:

Overall number of projects (#Proj) Large sample sizes allow for a larger variation in project properties and thus influence the conclusions which can be drawn from studies. Thus, we report the number of projects included in each dataset. This number does not include different releases or versions of a single project; these are counted as variants (see below).

Nature of projects (Syn?) We determined whether the projects used in the dataset have only been created for research purposes or whether they have been in actual use. While synthetic projects allow researchers to design projects with controlled properties, they are also a threat to the ecological validity of an evaluation. In contrast, projects which have been in actual use are more difficult to obtain and might vary widely in their properties.

- **synthetic (○)** Synthetic projects have been created for other purposes than actual usage, this also includes projects resulting from university courses.
- **real (empty cell)** Real projects are projects which have been created for actual usage. They do not have to be in use anymore. Further, this does not entail that the project evolved over a longer period.

Origin of tests As the evaluation of RTP approaches always relies on pools of test cases, we determine how the tests being used have been created. While manually created tests may be equivalent to historic tests in some scenarios, studies on historic test cases have a higher ecological validity.

- **historic (🕑)** The project includes tests created as part of the evolution of the project
- **manual (✎)** The tests were created manually, but not during the evolution of the project.
- **generated (⚙)** An automatic test generation tool was used to generate the tests.

Origin of variants Some RTP approaches require several variants of a program to gather data used in the prioritization. For example, many prioritization techniques require several faulty versions, and history-based techniques require a change history.

- **historic (🕑)** The variants used in the study are the result of the actual evolution of the project. This includes variants on several levels such as releases, snapshots, as well as commits.
- **manual mutants (✎)** Different variants of the project source code were created manually, but not as part of the development of the project.
- **generated mutants (⚙)** Some mutant generation tool was used to automatically generate mutants.

Origin of faults Previous work has shown, that for some prioritization techniques mutation faults can be used as a replacement for real, historic faults [19]. At the same time, manually seeded or generated faults pose a threat to the external validity of a study, as the historic set of fault of real-world projects might have extreme properties.

- **historic (🕑)** The faults are a result of the evolution of the project. This also includes studies in which the test suite of a newer version of a project is run on an older version.
- **manual (✎)** The faults were manually seeded.
- **generated (⚙)** The faults were generated.

Table 1: of all publications used in the literature survey described in section 2. A detailed description of the columns and symbols (resp. the dimensions and codes) can be found in section 2.2.

Key	Year	#Proj	Syn?	Tests	Variants	Faults	Dataset	C	V	T	R	Archived
[64]	2014	4		↻,✎	✎,⚙	⚙	SIR$_{java}$ (1)					
[74]	2008	2		↻	✎,↻	✎	SIR$_{java}$ (2)	✓	✓	✓		✓
[39]	2013	12		⚙,↻,✎	✎,↻	↻,✎	SIR$_{Siemens}$, SIR$_{java}$ (4), SIR$_{Space}$	✓		✓		
[45]	2010	1		✎	↻	✎	SIR$_{GNU}$ (1)	✓	✓	✓		
[48]	2006	7		✎	✎	✎	SIR$_{Siemens}$					
[73]	2013	1	○	✎	✎	✎						
[59]	2010	8		⚙,✎	✎	↻,✎	SIR$_{Space}$, SIR$_{Siemens}$	✓	✓	✓		✓
[2]	2013	1		↻	✎	✎		✓		✓		
[25]	2001	1		⚙,✎	✎	↻	space	✓	✓	✓		
[41]	2016	5		✎	↻	⚙	SIR$_{GNU}$ (5)	✓		✓		
[58]	2002	8		⚙,✎	✎	↻,✎	siemens, space					
[68]	2018	2		↻	↻	↻	GSDTSR	~	~	~	✓	~
[71]	2016	8		↻	↻	⚙		✓		✓		
[83]	2018	10		↻,✎	✎,↻	↻,✎	SIR$_{GNU}$ (5), defects4j (5)	✓	✓	✓	~	
[94]	2015	8		↻	↻	↻	SIR$_{java}$ (2)	✓	✓	✓		
[107]	2008	1		⚙	∅	?		✓	∅			
[115]	2016	1		↻,✎	∅	?			∅			
[119]	2012	3		↻,✎	✎,↻	✎,?	SIR$_{java}$ (1)			✓		
[125]	2013	4		↻	✎,↻,⚙	⚙,✎	SIR$_{java}$ (4)	✓		✓		
[6]	2012	1		↻	↻	↻	WebKit	✓	✓	✓		✓
[10]	2011	1		↻	↻	↻						
[17]	2005	4		↻	✎,↻,⚙	✎	SIR$_{java}$ (4)	✓		✓		
[44]	2008	1	○	⚙	✎	✎						
[61]	2008	6	~	⚙,✎	✎	✎	SIR$_{Siemens}$ (1)					
[62]	2005	3	○	⚙	✎	✎						
[79]	2013	1		↻	↻	↻						
[92]	1999	7		✎	✎	✎	siemens					
[111]	2006	1		⚙	✎	↻	SIR$_{Space}$	✓	✓	✓		✓
[127]	2009	2		↻	✎,↻	✎	SIR$_{java}$ (2)	✓	✓	✓		✓
[3]	2013	2		?	↻,⚙	⚙						
[27]	2011	1		↻	↻	?						
[60]	2009	7	~	⚙,✎	✎	✎	SIR$_{Siemens}$ (1)					
[77]	2012	1		↻	⚙	✎						
[84]	2013	1		↻,✎	↻	↻						
[91]	2013	3		✎	✎,↻,⚙	⚙,✎	SIR$_{GNU}$ (3)	✓		✓		
[95]	2008	3		✎	✎	✎		✓				
[98]	2010	1		✎	✎	✎	SIR$_{Siemens}$ (1)	✓	✓	✓		✓
[12]	2016	5		⚙	∅	⚙		✓	∅			
[56]	2012	10		↻	↻,⚙	⚙,↻		✓		✓		
[70]	2015	3		↻	↻	↻		✓		✓		
[85]	2015	5		↻	↻	↻	defects4j	✓	✓	✓		✓
[97]	2007	1		↻	↻	↻						
[106]	2012	3	○	⚙	⚙	⚙		✓				
[108]	2016	1		↻	↻	↻						
[20]	2008	5		↻	✎,⚙	⚙,✎	SIR$_{java}$ (5)	✓	✓	✓		✓
[28]	2015	6		↻,✎	✎,↻	↻,✎	SIR$_{GNU}$ (5)	✓	✓	✓		
[81]	2009	2		↻	✎,↻	↻,✎	SIR$_{java}$ (1)					
[101]	2017	3		↻	↻	↻					✓	✓
[104]	2002	3		↻	↻	∅						
[113]	2006	2	~	↻,✎	⚙	⚙		~		~		
[120]	2009	7		⚙,✎	✎,↻	↻,✎	SIR$_{Siemens}$ (2), SIR$_{Space}$, SIR$_{GNU}$ (2)	✓	✓	✓		✓
[124]	2014	6		⚙,✎	✎	↻,✎	SIR$_{GNU}$ (3), SIR$_{Space}$, SIR (1)	✓				
[126]	2009	2		↻	✎,↻	⚙,✎	SIR$_{java}$ (1)	✓		✓		
[53]	2009	11		✎	✎	✎	SIR$_{Siemens}$, SIR$_{GNU}$ (4)	✓		✓		
[65]	2009	7		✎	✎	✎	siemens					
[99]	2007	6	?	?	?	?						
[24]	2001	8		⚙,✎	✎	↻,✎	siemens, space					
[52]	2015	4		✎	✎,↻	✎	SIR$_{GNU}$ (4)	✓	✓	✓		✓
[78]	2015	3		↻	↻	↻						
[129]	2016	11		✎	✎,⚙	⚙,✎	SIR$_{Siemens}$	✓		✓		
[50]	2010	7		✎	✎	✎	SIR$_{Siemens}$	✓	✓	✓		✓
[4]	2018	3		↻	↻	↻,✎		✓	✓	✓		

Key	Year	#Proj	Syn?	Tests	Variants	Faults	Dataset	C	V	T	R	Archived
[57]	2017	2		🌀	🌀	🌀		✓	~	~		
[100]	2009	8	?	🌀	?	?						
[122]	2011	8		⚙,✎	✎,🌀,⚙	⚙,🌀,✎	SIR_{Space}, $SIR_{Siemens}$	✓		✓		
[14]	2012	2		🌀	🌀	🌀						
[114]	2015	4		✎	✎,🌀	✎	$SIR_{Siemens}$ (2), SIR_{GNU} (2)	✓	✓	✓		✓
[128]	2014	1		⚙,🌀	∅	🌀		✓	∅	✓		
[9]	2016	1		🌀	🌀	🌀						
[11]*a*	2018	50		🌀	🌀	⚙		✓	✓	✓		
[11]*a*	2018	11		🌀	🌀	🌀						
[18]	2006	5		🌀	✎,🌀,⚙	⚙,✎	SIR_{java} (4)	✓		✓		
[72]	2016	30		🌀	⚙	⚙		✓		✓		
[116]	2017	7		🌀	🌀,⚙	⚙,🌀		✓		✓		
[7]	2015	1		🌀	🌀	🌀		✓	✓	✓		
[67]*a*	2013	8		⚙,🌀,✎	?	?	$SIR_{Siemens}$ (6), SIR_{Space}, SIR_{GNU} (1)	✓	?	✓		
[67]*b*	2013	1		🌀	?	?		~	~	~		
[86]	2015	6		✎	✎,🌀	✎	$SIR_{Siemens}$ (2), SIR_{GNU} (4)	✓	✓	✓		
[123]	2015	4		🌀,✎	?	?		✓	?	✓		
[90]	2008	1		🌀	✎	✎	SIR_{java} (1)	✓	✓	✓		✓
[30]	2009	8		⚙,✎	✎	🌀,✎	SIR_{Space}, $SIR_{Siemens}$	✓	✓	✓		✓
[75]	2010	5		⚙,✎	✎	🌀,✎	$SIR_{Siemens}$ (4), SIR_{Space}	✓	✓	✓		
[87]	2016	15	~	🌀,✎	✎,🌀,⚙	⚙,✎	SIR (9), SIR_{java} (1)	✓		✓		
[66]	2012	7		✎	✎	✎	$SIR_{Siemens}$	✓	✓	✓		✓
[21]	2006	4		🌀	🌀	✎	SIR_{java} (4)	✓		✓		
[110]	2014	2		🌀	✎,🌀	✎	SIR_{java} (1), SIR (1)	✓	✓	✓		✓
[40]	2016	1	○	✎	∅	∅		✓	∅			
[63]*a*	2009	5	○	✎	🌀	🌀						
[63]*b*	2009	2		✎	✎	✎						
[63]*c*	2009	2		🌀	∅	🌀						
[80]	2011	1	○	✎	✎	✎						
[121]	2011	1		✎	✎	✎	siemens (1)					
[42]	2016	2		🌀	🌀	⚙						
[13]	2018	7	~	⚙	✎	✎						
[35]	2018	6		🌀	🌀	🌀	travistorrent	✓	✓	✓	✓	
[46]	2012	2		✎	✎,🌀	✎	SIR_{GNU} (2)	✓	✓	✓		✓
[49]	2008	7		✎	✎	✎	$SIR_{Siemens}$	✓	✓	✓		✓
[51]	2015	4		✎	✎,🌀	✎	SIR_{GNU} (4)	✓	✓	✓		✓
[89]	2016	1		🌀	🌀	🌀		✓	✓	✓		
[96]	2016	5		🌀	✎,🌀,⚙	⚙,✎	SIR_{java} (3), SIR (2)	✓		✓		
[102]*a*	2012	4	○	✎	✎	✎						
[102]*b*	2012	3		🌀	∅	🌀			∅			
[117]	2018	4		🌀,✎	✎,🌀	🌀,✎	SIR_{java} (2)	✓	✓	✓		✓
[31]	2013	2		⚙,🌀	🌀	✎		~				
[76]	2013	1	○	✎	✎	∅	triangle	✓				
[88]	2010	4	○	✎	✎	✎						
[105]	2008	1	○	✎	∅	?		✓	∅	✓		✓
[32]	2011	11		⚙,✎	✎,🌀	🌀,✎	$SIR_{Siemens}$, SIR_{Space}, SIR_{GNU} (3)	✓	✓	✓		✓
[29]	2014	5		🌀	⚙	⚙	SIR (4)	✓		✓		
[1]*a*	2016	3		🌀	✎	✎		✓		✓		
[1]*b*	2016	26	○	⚙	✎	✎						
[103]	2014	4	○	✎	✎	✎						
[109]	2012	5	○	✎	✎	✎		✓				
[38]	2014	33		⚙,🌀	🌀,⚙	⚙	SIR_{java} (4)	✓		~		
[16]	2010	5		🌀	✎,🌀	✎	SIR_{java} (5)	✓	✓	✓		✓
[19]*a*	2006	4		🌀	✎,⚙	⚙,✎	SIR_{java}(4)	✓		✓		
[19]*b*	2006	2		🌀	✎,⚙	⚙,✎	SIR_{java} (2)	✓		✓		
[22]	2016	6		🌀	✎,🌀,⚙	⚙,✎	SIR_{java} (6)	✓		✓		
[26]*a*	2002	8		⚙,✎	✎,🌀,⚙	🌀,✎	siemens, space					
[26]*b*	2002	3		🌀,✎	✎,🌀	🌀,✎		~	~	~		~
[36]	2013	6		🌀,✎	✎,🌀	🌀,✎		~		~		
[37]*a*	2016	10		⚙,🌀,✎	✎,🌀,⚙	⚙,🌀,✎	SIR_{java} (2), $SIR_{Siemens}$, SIR_{Space}	✓		✓		
[37]*b*	2016	5		?	🌀,⚙	⚙						
[54]	2003	1		⚙,✎	✎	🌀	space					
[82]	2012	4		🌀	✎,🌀,⚙	✎	SIR_{java} (4)	✓		✓		
[93]	2001	8		⚙,✎	✎,🌀,⚙	⚙,🌀,✎	siemens, space	✓	~	✓		~

Research Question 3. In order to approach the third research question, we characterized to which degree the datasets used in the studies are available. We distinguished between the general availability of parts of the data set (code, variants, tests, test runs) and whether the dataset was archived (archived):

Code (C) Is the source code of the project generally available? This is particularly challenging for studies using industrial projects or synthetic projects. Open-source projects used in studies are often available by their very nature.

Variants (V) Is a set of variants of the project available that includes the variants used in the project?

Tests (T) Are the tests available?

Test runs (R) Are the test runs and the corresponding results available? This is relevant for some history-based techniques.

Archived We consider the data archived if the full data as it was used can be retrieved from a website. If a study reuses an existing dataset, and this original dataset was archived (for example a SIR project), we count the dataset as archived only if the material was used without further modification such as the generation of mutants. In the special case of a study using randomly generated test suites from an archived pool of test cases, we still considered the data as being archived.

General Codes. We further used three generic codes throughout all dimensions:

unspecified (?) The information is, to our knowledge, not explicitly disclosed in the publication.

none (∅) There is none of what the dimension describes. For example, the study did not involve any variants of programs.

partially (~) The study falls in-between two binary codes.

2.3 Discussion

Based on the collected data (see table 1), we discuss general insights with respect to our three research questions.

Research Question 1. While some studies use datasets containing as many as 50 projects, the mean number of projects used per study is 5.25 ($SD = 6.36$). This might pose a threat to the external validity of some of the studies, as the variation in project properties might be smaller than what is to be expected from real-world settings.

Overall, the results show that only few studies incorporate synthetic programs (15.1%). While most studies did primarily use real projects, 21.4% of all studies used the *Siemens programs* [47]. While these are not synthetic, the *Siemens programs* are limited, as they are shorter than 1 000 LOC [15] [3].

With regard to the origin of the test cases, historic (54.8%) and manual test case pools (47.6%) are most often used in studies. Pools of generated test cases are only seldom used (23.0%). Notably, 26.2% of all studies mixed projects with test cases of different origin, although this might introduce a source of bias due to the different properties of these test case pools.

The variants used in the studies are also mostly historic (51.6%) and manual (56.3%). However, many of the historic variants are at the granularity of releases. For example, all projects with historic variants retrieved from SIR contain only releases as historic variants.

Finally, most projects contain manually seeded faults (57.9%), followed by historic faults (38.1%), and generated faults (22.2%). While this indicates that historic faults are often explicitly considered in studies, the percentage might also be a result of the fact that many projects were retrieved through SIR (see below), which provides some datasets which incorporate historic faults.

Research Question 2. We identified several datasets reused by studies on RTP. Many of these (42.9%) were retrieved through the Software-artifact Infrastructure Repository (SIR) [15]. Since 2015, studies also incorporated datasets from defects4j [55], travistorrent [5], and the Google Shared Dataset of Test Suite Results (GSDTSR) [23]. The most notable datasets are the the *Siemens programs* [47], the *space program* [112, 118], the *SIR Java dataset* [15], and the *SIR GNU dataset* [15]. The *siemens* and the *space* datasets were also used before they were made available through SIR.

Research Question 3. For many studies, in particular the ones using open-source projects, the basic data for the study can be considered available. At the same time, we can observe that datasets have seldom been properly archived. Only 19.0% of the datasets in the corpus are available through some form of archive (even given our wide definition of archiving). Further, most of these archived datasets (83.3%) are only considered archived because the described study used an unaltered dataset from SIR.

2.4 Summary of State of the Art of Datasets for Test Prioritization

SIR datasets are commonly used to evaluate RTP techniques and only few other datasets are used or made available. Even with the SIR datasets as a solid foundation for comparable experiments, the number of projects per dataset remains limited. Further, most of the test case pools, variants, and faults are manually created instead of relying on historic, real-world data. Finally, the *Siemens* dataset is commonly used, but the programs themselves are quite limited and do not resemble present-day software projects [15].

3 DATASET

To improve the ecological validity of future RTP evaluations, we propose to focus on real-world testing data from readily available software projects. As a first step, we construct a new dataset of open-source software projects and their fine-grained build data.

3.1 Projects

The RTPTorrent dataset contains 20 Java projects (listed in Table 2). All of them are available on GitHub and have been using the *TravisCI* build service. That means, both program evolution and test runs are available and analyzable using the repository itself, the *GHTorrent* dataset, and the *TravisTorrent* dataset [5, 33]. All test results are within the time range covered by *TravisTorrent* (2007 – 2016).

Our selection requirements were that the projects (1.) be written in Java due to frequent use in literature and large ecosystem of analysis and instrumentation tools, (2.) have the highest-ranking

[3] We classified the *Siemens programs* conservatively as real programs as the original paper only describes them as being "obtained from various sources" [47].

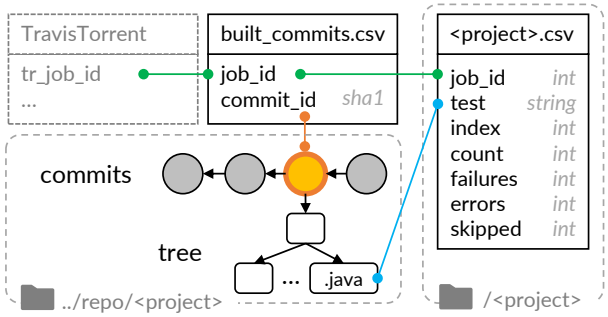


Figure 1: Relational structure of the dataset with links to TravisTorrent and the corresponding Git repository.

number of logged failures, and (3.) vary sufficiently in size and maturity to represent a broad spectrum of GitHub's community.

3.2 Structure

The dataset contains test results in relational form (see Figure 1) and the full source code history as Git Version Control System (Git) repositories.

Test results are available per *build job*. Builds run frequently during program evolution, but can span more than a single Git commit and spawn more than a single job (e.g. one per platform).

For each *build job* and each *test case* run during that build, we provide the number of total, failed, errored, and skipped test methods[4]. To facilitate use in prioritization, we also provide the index, which is the position in which the test case was originally run, and the duration as logged by *JUnit*. Note that durations below Java's clock resolution are reported as 0.0 s (9.28 % of all test cases) and 17 negative durations occurred due to defects in the test runner.

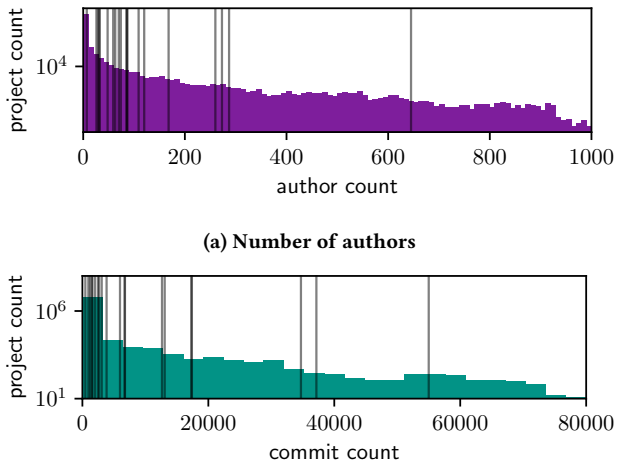
Dataset Compatibility. We aim to make our dataset compatible to the existing datasets *GHTorrent* and *TravisTorrent* without redundantly mirroring their data. That means:

- Our job and build IDs in the dataset refer to the TravisTorrent table (`travistorrent_8_2_2017.csv`), so that additional build information (e.g. branch, timestamps) can be obtained by joining.
- Our reported SHA1 hashes from Git commits are included in the `commits` table of GHTorrent. Authors, GitHub projects, associated pull requests and issues can be linked through this connection.

Due to the fact that we constructed the dataset based on build logs from 2007 to 2016, there are two limitations when linking to external resources:

- Not all projects are available on GitHub at their original locations. Due to re-engineering efforts, some repositories are merged into new projects that replace the original location. Since multiple forks of each project exist, we were able to archive the original repository from such a forked location and verify that our build commits are included.

[4]Method-level results are usually not logged on TravisCI



(a) Number of authors

(b) Number of commits

Figure 2: Distribution of the number of authors and commits over all Java projects on GitHub; black lines represent the projects of our dataset.

- Not all commits are available in the Git repositories, since proposed changes by contributors (pull requests, issue commits) can be built and tested but rejected by code reviewers. This affects 41 commits. The other 62 133 are part of the repositories.

3.3 Project Characteristics

Size. We quantified the size of our projects by structural and process metrics in Table 2. The number of lines and classes is computed for all Java code files in the latest build. The number of Git commits is the union over all commits built in the dataset, the sparkline illustrates their distribution over 9 years. We further state how many build jobs our dataset contains per project and the average number of test cases, test methods, and failing test methods per job.

Project Maturity and Representativeness. We compare our dataset to all Java projects in the *GHTorrent*[5] dataset to illustrate relative maturity and project size in relation to GitHub. Our projects have received on average 11 324 commits by 128 authors, while all Java projects on GitHub received on average 141 commits by 5 authors. How the number of commits and authors are distributed with respect to the background population can be seen in Figure 2. As such, our dataset is biased towards larger and contribution-heavy projects, which is acceptable, since test prioritization is of little benefit in smaller or less mature projects.

Reliability. Our dataset contains observations from build logs. Changes in the logging process or build configuration can alter how tests are parsed, grouped, or ordered, which makes test runs before and after such a change not directly comparable. In addition, there might be multiple types of builds configured, such as different platforms or different branches. They interleave since there is no

[5]Retrieved 2019-06-31

Table 2: Project selection with size (computed over the most recently built version), commit activity (2007 – 2016), build jobs, the average number of test cases (TC) and methods (TM) run and failing per build, and the average number of seconds (TC Time) each test case was running

Project	Lines	Classes	Activity	Commits	Builds	TC	TM	TM Failing	TC Time
Achilles	54 223	435		763	997	177.9	1373.41	5.46	0.6
DSpace	384 448	2025		3779	3338	62.94	708.18	2.37	2.1
HikariCP	13 868	69		1703	1662	28.28	117.59	0.57	2.0
LittleProxy	13 823	87		611	581	27.19	116.51	1.87	4.5
buck	562 536	3593		7147	1148	682.92	4052.42	3.48	2.1
cloudify	132 574	1024		11 078	5206	56.69	195.65	0.29	1.5
deeplearning4j	138 155	975		2607	1038	14.94	30.82	1.16	36.4
dynjs	57 184	724		531	1020	73.28	844.35	5.23	0.4
graylog2-server	127 161	1259		6136	10 622	129.33	943.18	0.1	5.7
jOOQ	351 209	1411		2006	3245	25.69	381.85	0.34	0.9
jade4j	10 288	148		374	932	38.51	258.86	7.55	0.1
jcabi-github	64 551	454		753	3241	171.29	605.47	0.55	4.0
jetty.project	346 354	1744		205	383	167.4	1468.23	3.26	5.0
jsprit	59 581	427		308	1089	86.55	997.14	1.13	0.3
okhttp	69 090	266		2308	9772	42.46	1195.77	1.62	3.8
optiq	243 064	1029		846	1808	44.16	1417.58	0.7	47.3
sling	673 484	4966		13 763	8552	181.9	1010.76	7.38	4.7
sonarqube	661 490	5486		5244	53 307	321.44	1948.13	0.68	3.0
titan	59 626	534		679	1075	46.51	478.03	5.97	68.5
wicket-bootstrap	42 352	524		1292	1110	46.1	159.93	32.74	0.3

way to distinguish them at build log or test level without resorting to heuristics or parsing of build configurations.

Another major source of variability is the time over which builds were collected. For most projects multiple years are included in which they grew significantly. The nonlinearity of this evolution can be seen at the example of the *graylog2-server* in Figure 3: From 2014 on, the number of test methods steeply rises. A bifurcation in 2014 hints that a different build configuration runs significantly more tests than the other configuration. The temporary drop in test case classes in 2014, which did not affect the number of test methods, indicates major re-engineering activity.

Timing data collected by TravisCI can be used to estimate the effort of a single test, but the same test can be run on increasingly powerful hardware over time and compete with an arbitrary number of concurrent build jobs. These sources of variability need to be taken into account.

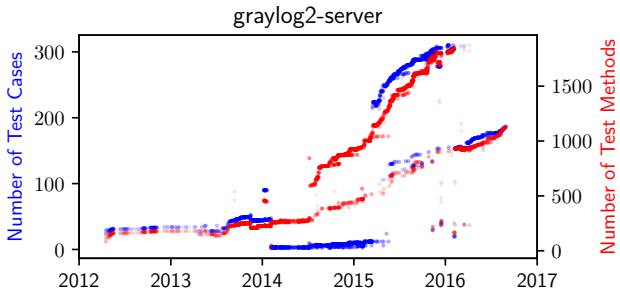


Figure 3: Number of test cases (blue/left) and test methods (red/right) per build in the *graylog2-server* project over time.

3.4 Data Procurement

Obtaining test-level data from build logs is challenging, as few tools continue running in the presence of failures and output sufficient information about succeeding tests. In TravisTorrent's raw build logs, we identified *Maven Surefire* and *Facebook Buck* as producing the most usable output that always logs results from all test cases, including the number of total, failing, erroring, and skipped test methods with timing information.

An example of a *Maven* build log is given in Listing 1. We used regular expressions to match which test is running and the failure and timing statistics. As a safety guard, we detect component boundaries (lines of minuses) and, within each component, pair corresponding test announcements with their results on a best-effort basis, as parallelization can announce multiple tests and then report all their results in the order they launched.

4 DEMONSTRATION AND BASELINE

Our dataset exhibits properties that are absent from synthetic datasets or projects observed only in large change increments. The fine-grained historical structure of our dataset reflects how tests respond to smaller changes and how real-world infrastructure is being used, including (but not limited to) the following factors:

- manually triggered builds
- building external contributions (pull requests) before accepting them

Listing 1: Excerpt from build log 56082549 of sonarqube, matched patterns highlighted.

```
...
[INFO] -------------------------------------------------------------------------
[INFO] Building SonarQube :: Server 5.2-SNAPSHOT
[INFO] -------------------------------------------------------------------------
...
Running org.sonar.server.es.request.ProxyDeleteRequestBuilderTest
Tests run: 6, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.285 sec
  - in org.sonar.server.es.request.ProxyDeleteRequestBuilderTest
Running org.sonar.server.search.QueryContextTest
Tests run: 12, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.016 sec
  - in org.sonar.server.search.QueryContextTest
Running org.sonar.server.activity.index.ActivityResultSetIteratorTest
Tests run: 3, Failures: 2, Errors: 0, Skipped: 0, Time elapsed: 0.197 sec <<< FAILURE!
  - in org.sonar.server.activity.index.ActivityResultSetIteratorTest
traverse_after_date(org.sonar.server.activity.index.ActivityResultSetIteratorTest)  Time elapsed: 0.006 sec
<<< FAILURE!
org.junit.ComparisonFailure: expected:<14200[668]00000L> but was:<14200[704]00000L>
at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
at sun.reflect.NativeConstructorAccessorImpl.newInstance(NativeConstructorAccessorImpl.java:62)
at sun.reflect.DelegatingConstructorAccessorImpl.newInstance(DelegatingConstructorAccessorImpl.java:45)
...
```

- streaks of repeatedly failing tests
- influence from configuration changes

We expect these phenomena to introduce noise and confounding factors, but they have the potential to act as predictors as well. To show the surprising effectiveness of using real-world patterns, we studied the heuristic of prioritizing tests by how recently they failed, also known as *demonstrated fault effectiveness* [58].

The purpose of this micro-study is to demonstrate the dataset in action and provide a baseline ranking. The baseline makes use of our fine-grained build history without being too complex and provides a non-trivial reference point other than *random* or *unmodified* test schedules.

Demonstrated Fault Effectiveness. This heuristic assigns a priority $P_t(n)$ to test t in build number n as $P_t(n) = \alpha F_t(n) + (1-\alpha)P_t(n-1)$ with $P(0) = 0$, where $F_t(n) = 1$ if the test t failed in build n, 0 otherwise. We only include past builds that were not running concurrently and fixed $\alpha = 0.8$ to focus on recent failures.

Measurements. The de-facto standard for evaluating test schedules is the Average Percentage of Faults Detected (APFD) metric. It measures how early faults are discovered. Given a test sequence T and a set of faults F where a fault f is detected after $TF(f)$ tests:

$$APFD(S,F) = 1 - \frac{\sum_{f \in F} TF(f)}{|S| \times |F|} + \frac{1}{2|S|} \quad (1)$$

Higher values correspond to earlier fault detection. If faults are not synthesized, a test failure is often equated with a fault, since telling apart distinct faults in real builds is hard to automate.

We quantified the APFD in our unprioritized dataset and additionaly provide a test schedule sorted by demonstrated fault effectiveness to serve as baseline. Table 3 shows the average results and their variability on a per-project basis and over all build jobs

Discussion. Although comparison with a wider range of existing prioritization strategies is out of scope, we observe that the simple heuristic provides competitive ranking performance. We attribute this property to the high resolution of our dataset, since we neither considered code nor change-related data as most prioritization techniques do. We encourage researchers to compare both history-aware and -oblivious heuristics when using such a dataset, as observing human programming activities can sometimes be a predictor that complements or outperforms formal relations between test suites and code.

Table 3: APFD scores of the original builds and the demonstrated fault effectiveness baseline, higher values are better. Variability is given as histogram from 0 % to 100 %, density to the right is better.

	Build APFD		Prioritized APFD	
Project	Mean [%]	Dist.	Mean [%]	Dist.
Achilles	29.0		53.0	
DSpace	34.7		80.9	
HikariCP	58.5		69.0	
LittleProxy	42.2		61.1	
buck	50.0		96.5	
cloudify	17.7		91.2	
deeplearning4j	48.3		83.4	
dynjs	45.7		58.2	
graylog2-server	45.4		72.8	
jOOQ	34.1		86.1	
jade4j	47.8		63.2	
jcabi-github	23.6		71.1	
jetty.project	20.3		86.1	
jsprit	46.7		62.0	
okhttp	45.7		79.5	
optiq	23.0		61.4	
sling	2.5		94.7	
sonarqube	36.4		74.4	
titan	27.5		75.0	
wicket-bootstrap	30.1		64.5	
Dataset	**25.9**		**81.1**	

5 CONCLUSION AND OUTLOOK

The evaluation of RTP techniques can be challenging with regard to balancing internal and external validity. The datasets on which RTP techniques are evaluated play an important role in the external validity of these studies.

Our literature survey showed that these evaluation studies reuse existing datasets. However, at the same time, many datasets include programs with limited scope and limited resemblance to real-world projects.

In response to that, we proposed a new dataset, based on 20 open-source Java projects constructed from data from GHTorrent and TravisTorrent. This dataset only consists of real-world data resulting from the evolution of the projects. To make this dataset accessible, we characterized the projects included in the dataset, illustrated how the dataset can be used, and which limitations and confounding factors researchers must expect. Finally, we provided a non-trivial baseline for future evaluations of RTP techniques by evaluating the peformance of the demonstrated fault effectiveness heuristic on our dataset.

As next steps, the current scope of 20 projects could be extended to be more representative, and a comparison with proprietary projects would be needed to asses the validity of open source findings in such settings. For future research, we hope that using such a dataset can uncover discrepancies between "clean-room" evaluations and the improvement they bring to real-world testing situations and inspire prioritization heuristics that better model the human nature of errors in development processes.

ACKNOWLEDGMENTS

This research is supported by the German Federal Ministry of Education and Research (BMBF) KI-LAB-ITSE grant, the HPI Research School for Service-oriented Systems Engineering, and the Hasso Plattner Design Thinking Research Program.

REFERENCES

[1] Everton L. G. Alves, Patrícia D. L. Machado, Tiago Massoni, and Miryung Kim. 2016. Prioritizing test cases for early detection of refactoring faults. *Softw. Test., Verif. Reliab.* 26, 5 (2016), 402–426.

[2] Everton L. G. Alves, Patrícia D. L. Machado, Tiago Massoni, and Samuel T. C. Santos. 2013. A refactoring-based approach for test case selection and prioritization. In *AST*. IEEE Computer Society, 93–99.

[3] Md. Junaid Arafeen and Hyunsook Do. 2013. Test Case Prioritization Using Requirements-Based Clustering. In *ICST*. IEEE Computer Society, 312–321.

[4] Maral Azizi and Hyunsook Do. 2018. A collaborative filtering recommender system for test case prioritization in web applications. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC 2018, Pau, France, April 09-13, 2018*. 1560–1567.

[5] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. TravisTorrent: synthesizing Travis CI and GitHub for full-stack research on continuous integration. In *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*. 447–450. https://doi.org/10.1109/MSR.2017.24

[6] Árpád Beszédes, Tamás Gergely, Lajos Schrettner, Judit Jász, Laszlo Lango, and Tibor Gyimóthy. 2012. Code coverage-based regression test selection and prioritization in WebKit. In *ICSM*. IEEE Computer Society, 46–55.

[7] Yi Bian, Serkan Kirbas, Mark Harman, Yue Jia, and Zheng Li. 2015. Regression Test Case Prioritisation for Guava. In *SSBSE (Lecture Notes in Computer Science)*, Vol. 9275. Springer, 221–227.

[8] Virginia Braun and Victoria Clarke. 2006. Using Thematic Analysis in Psychology. *Qualitative research in psychology* 3, 2 (2006), 77–101.

[9] Benjamin Busjaeger and Tao Xie. 2016. Learning for test prioritization: an industrial case study. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*. 975–980.

[10] Ryan Carlson, Hyunsook Do, and Anne Denton. 2011. A clustering approach to improving test case prioritization: An industrial case study. In *ICSM*. IEEE Computer Society, 382–391.

[11] Junjie Chen, Yiling Lou, Lingming Zhang, Jianyi Zhou, Xiaoleng Wang, Dan Hao, and Lu Zhang. 2018. Optimizing test prioritization via test distribution analysis. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. 656–667.

[12] Jinfu Chen, Lili Zhu, Tsong Yueh Chen, Rubing Huang, Dave Towey, Fei-Ching Kuo, and Yuchi Guo. 2016. An Adaptive Sequence Approach for OOS Test Case Prioritization. In *ISSRE Workshops*. IEEE Computer Society, 205–212.

[13] Jinfu Chen, Lili Zhu, Tsong Yueh Chen, Dave Towey, Fei-Ching Kuo, Rubing Huang, and Yuchi Guo. 2018. Test case prioritization for object-oriented software: An adaptive random sequence approach based on clustering. *Journal of Systems and Software* 135 (2018), 107–125.

[14] Pedro de Alcântara dos Santos Neto, Ricardo Britto, Thiago Soares, Werney Ayala, Jonathas Cruz, and Ricardo A. L. Rabêlo. 2012. Regression Testing Prioritization Based on Fuzzy Inference Systems. In *SEKE*. Knowledge Systems Institute Graduate School, 273–278.

[15] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. 2005. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empirical Software Engineering* 10, 4 (2005), 405–435. https://doi.org/10.1007/s10664-005-3861-2

[16] Hyunsook Do, Siavash Mirarab, Ladan Tahvildari, and Gregg Rothermel. 2010. The Effects of Time Constraints on Test Case Prioritization: A Series of Controlled Experiments. *IEEE Trans. Software Eng.* 36, 5 (2010), 593–617.

[17] Hyunsook Do and Gregg Rothermel. 2005. A Controlled Experiment Assessing Test Case Prioritization Techniques via Mutation Faults. In *ICSM*. IEEE Computer Society, 411–420.

[18] Hyunsook Do and Gregg Rothermel. 2006. An empirical study of regression testing techniques incorporating context and lifetime factors and improved cost-benefit models. In *SIGSOFT FSE*. ACM, 141–151.

[19] Hyunsook Do and Gregg Rothermel. 2006. On the Use of Mutation Faults in Empirical Assessments of Test Case Prioritization Techniques. *IEEE Trans. Software Eng.* 32, 9 (2006), 733–752.

[20] Hyunsook Do and Gregg Rothermel. 2008. Using sensitivity analysis to create simplified economic models for regression testing. In *ISSTA*. ACM, 51–62.

[21] Hyunsook Do, Gregg Rothermel, and Alex Kinneer. 2006. Prioritizing JUnit Test Cases: An Empirical Assessment and Cost-Benefits Analysis. *Empirical Software Engineering* 11, 1 (2006), 33–70.

[22] Sepehr Eghbali and Ladan Tahvildari. 2016. Test Case Prioritization Using Lexicographical Ordering. *IEEE Trans. Software Eng.* 42, 12 (2016), 1178–1195.

[23] Sebastian Elbaum, Andrew Mclaughlin, and John Penix. 2014. *The Google Dataset of Testing Results*. https://code.google.com/p/google-shared-dataset-of-test-suite-results

[24] Sebastian G. Elbaum, David Gable, and Gregg Rothermel. 2001. Understanding and Measuring the Sources of Variation in the Prioritization of Regression Test Suites. In *IEEE METRICS*. IEEE Computer Society, 169.

[25] Sebastian G. Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. 2001. Incorporating Varying Test Costs and Fault Severities into Test Case Prioritization. In *ICSE*. IEEE Computer Society, 329–338.

[26] Sebastian G. Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. 2002. Test Case Prioritization: A Family of Empirical Studies. *IEEE Trans. Software Eng.* 28, 2 (2002), 159–182.

[27] Emelie Engström, Per Runeson, and Andreas Ljung. 2011. Improving Regression Testing Transparency and Efficiency with History-Based Prioritization - An Industrial Case Study. In *ICST*. IEEE Computer Society, 367–376.

[28] Michael G. Epitropakis, Shin Yoo, Mark Harman, and Edmund K. Burke. 2015. Empirical evaluation of pareto efficient multi-objective regression test case prioritisation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*. 234–245.

[29] Chunrong Fang, Zhenyu Chen, Kun Wu, and Zhihong Zhao. 2014. Similarity-based test case prioritization using ordered sequences of program entities. *Software Quality Journal* 22, 2 (2014), 335–361.

[30] Yalda Fazlalizadeh, Alireza Khalilian, Mohammad Abdollahi Azgomi, and Saeed Parsa. 2009. Incorporating Historical Test Case Performance Data and Resource Constraints into Test Case Prioritization. In *Tests and Proofs, Third International Conference, TAP 2009, Zurich, Switzerland, July 2-3, 2009. Proceedings*. 43–57.

[31] Deepak Garg, Amitava Datta, and Tim French. 2013. A novel bipartite graph approach for selection and prioritisation of test cases. *ACM SIGSOFT Software Engineering Notes* 38, 6 (2013), 1–6.

[32] Alberto González-Sanchez, Éric Piel, Rui Abreu, Hans-Gerhard Groß, and Arjan J. C. van Gemund. 2011. Prioritizing tests for software fault diagnosis. *Softw., Pract. Exper.* 41, 10 (2011), 1105–1129.

[33] Georgios Gousios. 2013. The GHTorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories* (San Francisco,

CA, USA) *(MSR '13)*. IEEE Press, Piscataway, NJ, USA, 233–236. http://dl.acm.org/citation.cfm?id=2487085.2487132
[34] Maria Grant and Andrew Booth. 2009. A Typology of Reviews: An Analysis of 14 Review Types and Associated Methodologies. *Health Information & Libraries Journal* 26, 2 (2009), 91–108. https://doi.org/10.1111/j.1471-1842.2009.00848.x
[35] Alireza Haghighatkhah, Mika M"antyl"a, Markku Oivo, and Pasi Kuvaja. 2018. Test prioritization in continuous integration environments. *Journal of Systems and Software* 146 (2018), 80–98.
[36] Shifa-e-Zehra Haidry and Tim Miller. 2013. Using Dependency Structures for Prioritization of Functional Test Suites. *IEEE Trans. Software Eng.* 39, 2 (2013), 258–275.
[37] Dan Hao, Lu Zhang, Lei Zang, Yanbo Wang, Xingxia Wu, and Tao Xie. 2016. To Be Optimal or Not in Test-Case Prioritization. *IEEE Trans. Software Eng.* 42, 5 (2016), 490–504.
[38] Dan Hao, Lingming Zhang, Lu Zhang, Gregg Rothermel, and Hong Mei. 2014. A Unified Test Case Prioritization Approach. *ACM Trans. Softw. Eng. Methodol.* 24, 2 (2014), 10:1–10:31.
[39] Dan Hao, Xu Zhao, and Lu Zhang. 2013. Adaptive Test-Case Prioritization Guided by Output Inspection. In *COMPSAC*. IEEE Computer Society, 169–179.
[40] Ramzi A. Haraty, Nashat Mansour, Lama Moukahal, and Iman Khalil. 2016. Regression Test Cases Prioritization Using Clustering and Code Change Relevance. *International Journal of Software Engineering and Knowledge Engineering* 26, 5 (2016), 733–768.
[41] Christopher Henard, Mike Papadakis, Mark Harman, Yue Jia, and Yves Le Traon. 2016. Comparing white-box and black-box test prioritization. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. 523–534.
[42] Charitha Hettiarachchi, Hyunsook Do, and Byoungju Choi. 2016. Risk-based test case prioritization using a fuzzy expert system. *Information & Software Technology* 69 (2016), 1–15.
[43] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. 2017. Trade-offs in continuous integration: assurance, security, and flexibility. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 197–207.
[44] Shan-Shan Hou, Lu Zhang, Tao Xie, and Jiasu Sun. 2008. Quota-constrained test-case prioritization for regression testing of service-centric systems. In *ICSM*. IEEE Computer Society, 257–266.
[45] Yu-Chi Huang, Chin-Yu Huang, Jun-Ru Chang, and Tsan-Yuan Chen. 2010. Design and Analysis of Cost-Cognizant Test Case Prioritization Using Genetic Algorithm with Test History. In *COMPSAC*. IEEE Computer Society, 413–418.
[46] Yu-Chi Huang, Kuan-Li Peng, and Chin-Yu Huang. 2012. A history-based cost-cognizant test case prioritization technique in regression testing. *Journal of Systems and Software* 85, 3 (2012), 626–637.
[47] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. 1994. Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria. In *Proceedings of 16th International Conference on Software Engineering*. 191–200. https://doi.org/10.1109/ICSE.1994.296778
[48] Dennis Jeffrey and Neelam Gupta. 2006. Test Case Prioritization Using Relevant Slices. In *COMPSAC (1)*. IEEE Computer Society, 411–420.
[49] Dennis Jeffrey and Neelam Gupta. 2008. Experiments with test case prioritization using relevant slices. *Journal of Systems and Software* 81, 2 (2008), 196–221.
[50] Bo Jiang and W. K. Chan. 2010. On the Integration of Test Adequacy, Test Case Prioritization, and Statistical Fault Localization. In *QSIC*. IEEE Computer Society, 377–384.
[51] Bo Jiang and Wing Kwong Chan. 2015. Input-based adaptive randomized test case prioritization: A local beam search approach. *Journal of Systems and Software* 105 (2015), 91–106.
[52] Bo Jiang, Wing Kwong Chan, and T. H. Tse. 2015. PORA: Proportion-Oriented Randomized Algorithm for Test Case Prioritization. In *QRS*. IEEE, 131–140.
[53] Bo Jiang, Zhenyu Zhang, Wing Kwong Chan, and T. H. Tse. 2009. Adaptive Random Test Case Prioritization. In *ASE*. IEEE Computer Society, 233–244.
[54] James A. Jones and Mary Jean Harrold. 2003. Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage. *IEEE Trans. Software Eng.* 29, 3 (2003), 195–209.
[55] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*. 437–440. https://doi.org/10.1145/2610384.2628055
[56] René Just, Gregory M. Kapfhammer, and Franz Schweiggert. 2012. Using Non-redundant Mutation Operators and Test Suite Prioritization to Achieve Efficient and Scalable Mutation Analysis. In *ISSRE*. IEEE Computer Society, 11–20.
[57] Jeongho Kim, Hohyeon Jeong, and Eunseok Lee. 2017. Failure history data-based test case prioritization for effective regression test. In *Proceedings of the Symposium on Applied Computing, SAC 2017, Marrakech, Morocco, April 3-7, 2017*. 1409–1415.
[58] Jung-Min Kim and Adam A. Porter. 2002. A history-based test prioritization technique for regression testing in resource constrained environments. In *ICSE*. ACM, 119–129.
[59] Sejun Kim and Jongmoon Baik. 2010. An effective fault aware test case prioritization by incorporating a fault localization technique. In *ESEM*. ACM.
[60] Bogdan Korel and George Koutsogiannakis. 2009. Experimental Comparison of Code-Based and Model-Based Test Prioritization. In *ICST Workshops*. IEEE Computer Society, 77–84.
[61] Bogdan Korel, George Koutsogiannakis, and Luay Ho Tahat. 2008. Application of system models in regression test suite prioritization. In *ICSM*. IEEE Computer Society, 247–256.
[62] Bogdan Korel, Luay Ho Tahat, and Mark Harman. 2005. Test Prioritization Using System Models. In *ICSM*. IEEE Computer Society, 559–568.
[63] R. Krishnamoorthi and S. A. Sahaaya Arul Mary. 2009. Requirement Based System Test Case Prioritization of New and Regression Test Cases. *International Journal of Software Engineering and Knowledge Engineering* 19, 3 (2009), 453–475.
[64] Jung-Hyun Kwon, In-Young Ko, Gregg Rothermel, and Matt Staats. 2014. Test Case Prioritization Based on Information Retrieval Concepts. In *APSEC (1)*. IEEE Computer Society, 19–26.
[65] Yves Ledru, Alexandre Petrenko, and Sergiy Boroday. 2009. Using String Distances for Test Case Prioritisation. In *ASE*. IEEE Computer Society, 510–514.
[66] Yves Ledru, Alexandre Petrenko, Sergiy Boroday, and Nadine Mandran. 2012. Prioritizing test cases with string distances. *Autom. Softw. Eng.* 19, 1 (2012), 65–95.
[67] Zheng Li, Yi Bian, Ruilian Zhao, and Jun Cheng. 2013. A Fine-Grained Parallel Multi-objective Test Case Prioritization on GPU. In *SSBSE (Lecture Notes in Computer Science)*, Vol. 8084. Springer, 111–125.
[68] Jingjing Liang, Sebastian G. Elbaum, and Gregg Rothermel. 2018. Redefining prioritization: continuous prioritization for continuous integration. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. 688–698.
[69] Yiling Lou, Junjie Chen, Lingming Zhang, and Dan Hao. 2019. Chapter One - A Survey on Regression Test-Case Prioritization. In *Advances in Computers*, Atif M. Memon (Ed.). Vol. 113. Elsevier, 1–46. https://doi.org/10.1016/bs.adcom.2018.10.001
[70] Yiling Lou, Dan Hao, and Lu Zhang. 2015. Mutation-based test-case prioritization in software evolution. In *ISSRE*. IEEE Computer Society, 46–57.
[71] Yafeng Lu, Yiling Lou, Shiyang Cheng, Lingming Zhang, Dan Hao, Yangfan Zhou, and Lu Zhang. 2016. How does regression test prioritization perform in real-world software evolution?. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. 535–546.
[72] Qi Luo, Kevin Moran, and Denys Poshyvanyk. 2016. A large-scale empirical comparison of static and dynamic test case prioritization techniques. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*. 559–570.
[73] Junpeng Lv, Bei-Bei Yin, and Kai-Yuan Cai. 2013. On the Gain of Measuring Test Case Prioritization. In *COMPSAC*. IEEE Computer Society, 627–632.
[74] Zengkai Ma and Jianjun Zhao. 2008. Test Case Prioritization Based on Analysis of Program Structure. In *APSEC*. IEEE Computer Society, 471–478.
[75] Camila Loiola Brito Maia, Rafael Augusto Ferreira do Carmo, Fabricio Gomes de Freitas, Gustavo Augusto Lima de Campos, and Jerffeson Teixeira de Souza. 2010. Automated Test Case Prioritization with Reactive GRASP. *Adv. Software Engineering* 2010 (2010), 428521:1–428521:18.
[76] Ruchika Malhotra and Divya Tiwari. 2013. Development of a framework for test case prioritization using genetic algorithm. *ACM SIGSOFT Software Engineering Notes* 38, 3 (2013), 1–6.
[77] Christoph Malz, Nasser Jazdi, and Peter Göhner. 2012. Prioritization of Test Cases Using Software Agents and Fuzzy Logic. In *ICST*. IEEE Computer Society, 483–486.
[78] Dusica Marijan. 2015. Multi-perspective Regression Test Prioritization for Time-Constrained Environments. In *QRS*. IEEE, 157–162.
[79] Dusica Marijan, Arnaud Gotlieb, and Sagar Sen. 2013. Test Case Prioritization for Continuous Regression Testing: An Industrial Case Study. In *ICSM*. IEEE Computer Society, 540–543.
[80] S. A. Sahaaya Arul Mary and R. Krishnamoorthi. 2011. Time-Aware and Weighted Fault Severity Based Metrics for Test Case Prioritization. *International Journal of Software Engineering and Knowledge Engineering* 21, 1 (2011), 129–142.
[81] Wes Masri and Marwa El-Ghali. 2009. Test case filtering and prioritization based on coverage of combinations of program elements. In *WODA*. 29–34.
[82] Hong Mei, Dan Hao, Lingming Zhang, Lu Zhang, Ji Zhou, and Gregg Rothermel. 2012. A Static Approach to Prioritizing JUnit Test Cases. *IEEE Trans. Software Eng.* 38, 6 (2012), 1258–1275.
[83] Breno Miranda, Emilio Cruciani, Roberto Verdecchia, and Antonia Bertolino. 2018. FAST approaches to scalable similarity-based test case prioritization. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. 222–232.
[84] Daniel Di Nardo, Nadia Alshahwan, Lionel C. Briand, and Yvan Labiche. 2013. Coverage-Based Test Case Prioritisation: An Industrial Case Study. In *ICST*. IEEE Computer Society, 302–311.

[85] Tanzeem Bin Noor and Hadi Hemmati. 2015. A similarity-based approach for test case prioritization using historical failure data. In *ISSRE*. IEEE Computer Society, 58–68.

[86] Dario Di Nucci, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. 2015. Hypervolume-Based Search for Test Case Prioritization. In *SSBSE (Lecture Notes in Computer Science)*, Vol. 9275. Springer, 157–172.

[87] Subhrakanta Panda, Dishant Munjal, and Durga Prasad Mohapatra. 2016. A Slice-Based Change Impact Analysis for Regression Test Case Prioritization of Object-Oriented Programs. *Adv. Software Engineering* 2016 (2016), 7132404:1–7132404:20.

[88] Chhabi Rani Panigrahi and Rajib Mall. 2010. Model-based regression test case prioritization. *ACM SIGSOFT Software Engineering Notes* 35, 6 (2010), 1–7.

[89] Jos'e Antonio Parejo, Ana B. S'anchez, Sergio Segura, Antonio Ruiz Cort'es, Roberto E. Lopez-Herrejon, and Alexander Egyed. 2016. Multi-objective test case prioritization in highly configurable systems: A case study. *Journal of Systems and Software* 122 (2016), 287–310.

[90] Hyuncheol Park, Hoyeon Ryu, and Jongmoon Baik. 2008. Historical Value-Based Approach for Cost-Cognizant Test Case Prioritization to Improve the Effectiveness of Regression Testing. In *SSIRI*. IEEE Computer Society, 39–46.

[91] Xiao Qu and Myra B. Cohen. 2013. A Study in Prioritization for Higher Strength Combinatorial Testing. In *ICST Workshops*. IEEE Computer Society, 285–294.

[92] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. 1999. Test Case Prioritization: An Empirical Study. In *ICSM*. IEEE Computer Society, 179–188.

[93] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. 2001. Prioritizing Test Cases For Regression Testing. *IEEE Trans. Software Eng.* 27, 10 (2001), 929–948.

[94] Ripon K. Saha, Lingming Zhang, Sarfraz Khurshid, and Dewayne E. Perry. 2015. An Information Retrieval Approach for Regression Test Prioritization Based on Program Changes. In *ICSE (1)*. IEEE Computer Society, 268–279.

[95] Sreedevi Sampath, Renée C. Bryce, Gokulanand Viswanath, Vani Kandimalla, and Akif Günes Koru. 2008. Prioritizing User-Session-Based Test Cases for Web Applications Testing. In *ICST*. IEEE Computer Society, 141–150.

[96] Amanda Schwartz and Hyunsook Do. 2016. Cost-effective regression testing through Adaptive Test Prioritization strategies. *Journal of Systems and Software* 115 (2016), 61–81.

[97] Mark Sherriff, Mike Lake, and Laurie Williams. 2007. Prioritization of Regression Tests using Singular Value Decomposition with Empirical Change Records. In *ISSRE*. IEEE Computer Society, 81–90.

[98] Cristian Simons and Emerson Cabrera Paraiso. 2010. Regression test cases prioritization using Failure Pursuit Sampling. In *ISDA*. IEEE, 923–928.

[99] Adam M. Smith, Joshua Geiger, Gregory M. Kapfhammer, and Mary Lou Soffa. 2007. Test suite reduction and prioritization with call trees. In *ASE*. ACM, 539–540.

[100] Adam M. Smith and Gregory M. Kapfhammer. 2009. An empirical study of incorporating cost into test suite reduction and prioritization. In *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC), Honolulu, Hawaii, USA, March 9-12, 2009*. 461–467.

[101] Helge Spieker, Arnaud Gotlieb, Dusica Marijan, and Morten Mossige. 2017. Reinforcement learning for automatic test case prioritization and selection in continuous integration. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*. 12–22.

[102] Hema Srikanth and Sean Banerjee. 2012. Improving test efficiency through system test prioritization. *Journal of Systems and Software* 85, 5 (2012), 1176–1187.

[103] Hema Srikanth, Sean Banerjee, Laurie Williams, and Jason A. Osborne. 2014. Towards the prioritization of system test cases. *Softw. Test., Verif. Reliab.* 24, 4 (2014), 320–337.

[104] Amitabh Srivastava and Jay Thiagarajan. 2002. Effectively prioritizing tests in development environment. In *ISSTA*. ACM, 97–106.

[105] Praveen Ranjan Srivastava, Krishan Kumar, and G. Raghurama. 2008. Test case prioritization based on requirements and risk factors. *ACM SIGSOFT Software Engineering Notes* 33, 4 (2008).

[106] Matt Staats, Pablo Loyola, and Gregg Rothermel. 2012. Oracle-Centric Test Case Prioritization. In *ISSRE*. IEEE Computer Society, 311–320.

[107] Heiko Stallbaum, Andreas Metzger, and Klaus Pohl. 2008. An Automated Technique for Risk-based Test Case Generation and Prioritization. In *AST*. ACM, 67–70.

[108] Per Erik Strandberg, Daniel Sundmark, Wasif Afzal, Thomas J. Ostrand, and Elaine J. Weyuker. 2016. Experience Report: Automated System Level Regression Test Prioritization Using Multiple Factors. In *ISSRE*. IEEE Computer Society, 12–23.

[109] Luay Ho Tahat, Bogdan Korel, Mark Harman, and Hasan Ural. 2012. Regression test suite prioritization using system models. *Softw. Test., Verif. Reliab.* 22, 7 (2012), 481–506.

[110] Stephen W. Thomas, Hadi Hemmati, Ahmed E. Hassan, and Dorothea Blostein. 2014. Static test case prioritization using topic models. *Empirical Software Engineering* 19, 1 (2014), 182–212.

[111] Paolo Tonella, Paolo Avesani, and Angelo Susi. 2006. Using the Case-Based Ranking Methodology for Test Case Prioritization. In *ICSM*. IEEE Computer Society, 123–133.

[112] Filippos I. Vokolos and Phyllis G. Frankl. 1998. Empirical Evaluation of the Textual Differencing Regression Testing Technique. In *1998 International Conference on Software Maintenance, ICSM 1998, Bethesda, Maryland, USA, November 16-19, 1998*. 44–53. https://doi.org/10.1109/ICSM.1998.738488

[113] Kristen R. Walcott, Mary Lou Soffa, Gregory M. Kapfhammer, and Robert S. Roos. 2006. TimeAware test suite prioritization. In *ISSTA*. ACM, 1–12.

[114] Rongcun Wang, Shujuan Jiang, and Deng Chen. 2015. Similarity-based regression test case prioritization. In *SEKE*. KSI Research Inc. and Knowledge Systems Institute Graduate School, 358–363.

[115] Shuai Wang, Shaukat Ali, Tao Yue, Øyvind Bakkeli, and Marius Liaaen. 2016. Enhancing test case prioritization in an industrial setting with resource awareness and multi-objective search. In *ICSE (Companion Volume)*. ACM, 182–191.

[116] Song Wang, Jaechang Nam, and Lin Tan. 2017. QTEP: quality-aware test case prioritization. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*. 523–534.

[117] Ying Wang, Zhiliang Zhu, Bo Yang, Fangda Guo, and Hai Yu. 2018. Using reliability risk analysis to prioritize test cases. *Journal of Systems and Software* 139 (2018), 14–31.

[118] W. Eric Wong, Joseph Robert Horgan, Aditya P. Mathur, and Alberto Pasquini. 1997. Test Set Size Minimization and Fault Detection Effectiveness: A Case Study in a Space Application. In *COMPSAC*. IEEE Computer Society, 522–528.

[119] Kun Wu, Chunrong Fang, Zhenyu Chen, and Zhihong Zhao. 2012. Test case prioritization incorporating ordered sequence of program elements. In *AST*. IEEE Computer Society, 124–130.

[120] Shin Yoo, Mark Harman, Paolo Tonella, and Angelo Susi. 2009. Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009, Chicago, IL, USA, July 19-23, 2009*. 201–212.

[121] Hoijin Yoon and Byoungju Choi. 2011. A Test Case Prioritization Based on Degree of Risk Exposure and its Empirical Study. *International Journal of Software Engineering and Knowledge Engineering* 21, 2 (2011), 191–209.

[122] Dongjiang You, Zhenyu Chen, Baowen Xu, Bin Luo, and Chen Zhang. 2011. An empirical study on the effectiveness of time-aware test case prioritization techniques. In *Proceedings of the 2011 ACM Symposium on Applied Computing (SAC), TaiChung, Taiwan, March 21 - 24, 2011*. 1451–1456.

[123] Fang Yuan, Yi Bian, Zheng Li, and Ruilian Zhao. 2015. Epistatic Genetic Algorithm for Test Case Prioritization. In *SSBSE (Lecture Notes in Computer Science)*, Vol. 9275. Springer, 109–124.

[124] Chaoqiang Zhang, Alex Groce, and Mohammad Amin Alipour. 2014. Using test case reduction and prioritization to improve symbolic execution. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*. 160–170.

[125] Lingming Zhang, Dan Hao, Lu Zhang, Gregg Rothermel, and Hong Mei. 2013. Bridging the gap between the total and additional test-case prioritization strategies. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. 192–201.

[126] Lu Zhang, Shan-Shan Hou, Chao Guo, Tao Xie, and Hong Mei. 2009. Time-aware test-case prioritization using integer linear programming. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009, Chicago, IL, USA, July 19-23, 2009*. 213–224.

[127] Lingming Zhang, Ji Zhou, Dan Hao, Lu Zhang, and Hong Mei. 2009. Prioritizing JUnit test cases in absence of coverage information. In *ICSM*. IEEE Computer Society, 19–28.

[128] Xiaofang Zhang, Tsong Yueh Chen, and Huai Liu. 2014. An Application of Adaptive Random Sequence in Test Case Prioritization. In *SEKE*. Knowledge Systems Institute Graduate School, 126–131.

[129] Xiaofang Zhang, Xiaoyuan Xie, and Tsong Yueh Chen. 2016. Test Case Prioritization Using Adaptive Random Sequence with Category-Partition-Based Distance. In *QRS*. IEEE, 374–385.