Keeping Mutation Test Suites Consistent and Relevant with Long-Standing Mutants

Milos Ojdanic
milos.ojdanic@uni.lu
University of Luxembourg
Luxembourg

Mike Papadakis
michail.papadakis@uni.lu
University of Luxembourg
Luxembourg

Mark Harman
mark.harman@ucl.ac.uk
Meta platforms Inc. and UCL
UK

ABSTRACT

Mutation testing has been demonstrated to be one of the most powerful fault-revealing tools in the tester's tool kit. Much previous work implicitly assumed it to be sufficient to re-compute mutant suites per release. Sadly, this makes mutation results inconsistent; mutant scores from each release cannot be directly compared, making it harder to measure test improvement. Furthermore, regular code change means that a mutant suite's relevance will naturally degrade over time. We measure this degradation in relevance for 143,500 mutants in 4 non-trivial systems, finding that 52% degrade, on average. We introduce a mutant brittleness measure and use it to audit software systems and their mutation suites. We also demonstrate how consistent-by-construction long-standing mutant suites can be identified with a 10x improvement in mutant relevance over an arbitrary test suite. Our results indicate that the research community should avoid the re-computation of mutant suites and focus, instead, on long-standing mutants, thereby improving the consistency and relevance of mutation testing.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Evolving Systems, Mutation Testing, Test Adequacy, Continuous Integration, Software Testing

ACM Reference Format:
Milos Ojdanic, Mike Papadakis, and Mark Harman. 2023. Keeping Mutation Test Suites Consistent and Relevant with Long-Standing Mutants. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3611643.3613089

1 INTRODUCTION

Mutation Testing has been demonstrated to be one of the Software Testing community's most effective software testing techniques [6]. It seeds artificial faults - known as mutants. When a test distinguishes the behaviour of the mutant from the original, the mutant is said to be 'killed'. Test effectiveness is captured and represented by the mutation score - the proportion of mutants killed [15, 22, 28].

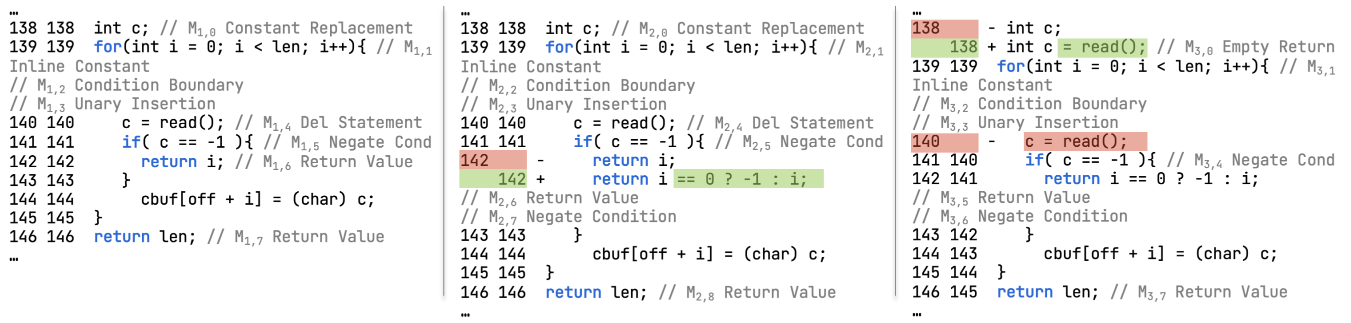
Initial barriers to adoption concerned the computational cost of the approach [13, 33]. Recent techniques have tackled the mutation cost problem using intelligent mutant selection [13, 33], higher order mutation [14, 25], commit awareness [23, 24] and mutant subsumption [2, 8, 31], thereby removing these cost-based barriers.

We argue that there is a remaining barrier to uptake: mutant consistency. We need a *consistent* set of mutants for a project so that test effectiveness can be consistently tracked against a common baseline over a series of project releases. Sadly, almost all existing research on mutation testing assumes that a fresh set of mutants is created for each release of the system [15, 28]. An alternative would be to fix a set of mutants as a baseline and use this to measure ongoing test effectiveness evolution. However, this paper shows, that such a fixed mutation set will quickly degrade in its ability to provide a measurement of test effectiveness relevance over time.

We introduce a mutation test brittleness metric, which can be used to assess a mutation suite, and software project, in terms of the rate at which mutant relevance decays over a series of releases, w.r.t., allowing the mutant suite to stand watchful of core test requirements between the releases. Our results demonstrate that mutants have diverse life spans across program versions. We show that a high-quality suite of *long-standing* mutants allows us to maintain effectiveness over a series of releases: a long-standing mutant suite provides test effectiveness relevance for at least 10x longer than a randomly selected suite. In order to increase the consistency and relevance of mutation testing towards continuous integration, we provide a vision and with promising preliminary results, we conclude that the research community should focus on long-standing mutants, their applications, the opportunities they open, and the remaining open questions.

Specifically, this paper's primary contributions are: ***(1)*** The introduction of long-standing mutants as an important category warranting further study in the context of continuous integration. ***(2)*** The introduction of metrics for assessing mutant brittleness and visualisations of how this metric varies for a given project over time. ***(3)*** An empirical study of long-standing mutants based on four non-trivial systems and 143,500 mutants. ***(4)*** The key motivating finding is that long-standing mutant suites enjoy an order of magnitude longer relevance than a randomly selected suite over the four systems studied. ***(5)*** An important 'special relationship' between long-standing and subsuming mutants: mutants that are subsuming in one version have a high probability of subsuming in the following versions. This relationship opens optimistic prospects for mutants' ability to maintain consistency, relevance, effectiveness *and* subsumption over a series of releases.

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0327-0/23/12.
https://doi.org/10.1145/3611643.3613089



```
…
138 138  int c; // M1,0 Constant Replacement
139 139  for(int i = 0; i < len; i++){ // M1,1
Inline Constant
// M1,2 Condition Boundary
// M1,3 Unary Insertion
140 140     c = read(); // M1,4 Del Statement
141 141     if( c == -1 ){ // M1,5 Negate Cond
142 142       return i; // M1,6 Return Value
143 143     }
144 144       cbuf[off + i] = (char) c;
145 145  }
146 146  return len; // M1,7 Return Value
…
```

```
…
138 138  int c; // M2,0 Constant Replacement
139 139  for(int i = 0; i < len; i++){ // M2,1
Inline Constant
// M2,2 Condition Boundary
// M2,3 Unary Insertion
140 140     c = read(); // M2,4 Del Statement
141 141     if( c == -1 ){ // M2,5 Negate Cond
142      -      return i;
    142  +      return i == 0 ? -1 : i;
// M2,6 Return Value
// M2,7 Negate Condition
143 143     }
144 144       cbuf[off + i] = (char) c;
145 145  }
146 146  return len; // M2,8 Return Value
…
```

```
…
138      - int c;
    138  + int c = read(); // M3,0 Empty Return
139 139  for(int i = 0; i < len; i++){ // M3,1
Inline Constant
// M3,2 Condition Boundary
// M3,3 Unary Insertion
140      -    c = read();
141 140     if( c == -1 ){ // M3,4 Negate Cond
142 141       return i == 0 ? -1 : i;
// M3,5 Return Value
// M3,6 Negate Condition
143 142     }
144 143       cbuf[off + i] = (char) c;
145 144  }
146 145  return len; // M3,7 Return Value
…
```

Figure 1: Example of mutants standing through 3 chronological sequences of code versions. The example code snippet comes from Apache commons-io project, while method read() is excerpted from the BoundedReader.java (versions around 81210eb). The green and red rectangles represent associated commit changes. While java comments (//) describe the set of mutants $M_{i,j}$, where i is the observed program version, and j is a mutant ID.

2 BACKGROUND AND RELATED WORK

Commit-Relevant Mutants. Applying traditional mutation testing in CI processes is impractical due to its cost. Meanwhile, Commit-Aware Mutation Testing scales and defines commit-relevant mutants as a set of mutants affected by the changed program behaviour that serve as commit-relevant test requirements to guide test assessment by aiming at the changed program functionality [4, 24]. Learning-based approaches [19] emerged capable of learning the commit-relevant mutants thus showing potential and opening a direction towards learning mutant's behaviour in evolving context. However - it is necessary to realise that to improve the testing process continuously and thus quantify overall testing quality - it would require applying the technique after each program change cycle to not indebt and lose test requirements. The merit of reappearing mature mutants is that they preserve test requirements and thus complement commit-relevant mutants. In particular, the long-standing mutants promise to keep overlooked testing requirements from oblivion and provide test assessment for a prolonged time.

Table 1: Observed files through projects evolution

Observed Files	Time points	Mutants	Commons Project
CSVParser	31	8757	csv
CSVRecord	16	2656	csv
Lexer	21	10688	csv
CSVLexer	17	11208	csv
CSVFormat	47	52432	csv
CSVPrinter	21	20382	csv
IterableUtils	10	6441	collections
CharSequenceUtils	10	6802	lang
WordUtils	15	24128	text

Subsuming Mutants. In an attempt to further scale and make test assessment affordable, many recent studies (consult the survey by Papadakis et al. [26–28]) consider subsuming mutants to reduce the number of mutants required to measure test adequacy [28]. Specifically, a subsumption relationship between mutants emerges from mutant behaviours, implying that the majority of the mutants fall into the redundancy basket since distinguishing subsuming mutants will lead to the identification of all other mutants [16].

More formally, given a finite set of mutants M and a finite set of tests T, mutant m_i is said to dynamically subsume mutant m_j if every test in T that kills m_i also kills m_j [2]. Calculating test effectiveness over subsuming mutants offers a much better indicator than the traditional mutation score since subsuming mutants have an almost linear relationship between the number of tests, providing more practicality for determining how much testing work remains over how much has been completed [17].

Although the evidence is strong and the benefits are multi-fold, calculating subsumption in real time requires knowledge of mutant's behaviour, usually represented through test execution, which is unpractical in real-time. Knowing whether the mutants can carry the subsumption information from version to version promises insights into their redundancy, cost utilization and testing investment - priority. Recently few approaches have used learning-based methods to target subsuming mutants with a certain level of confidence, considering their location and properties [8]. In this paper, we reuse the guarantee of the quality of test assessment when the mutants are *also* long-standing.

3 LONG-STANDING MUTANTS

3.1 Motivating Example

A key challenge in the current state of mutation regression testing is a sequential version to version execution. Suppose we keep a record of generated mutants to the same element on which the mutant is generated and version them from one version to another. Figure 1, depicts a chronological sequence of 3 different versions of method `read()` extracted from Apache Commons-io project. Following the evolution of the code, we can observe the evolution of the mutant set. We notice that most mutants reside in the same place across the versions. When a mutant location is unchanged from version to version, we consider that it stands through time. While the number of versions the mutant occurs on the same element is continuous, it promises to serve as a priority weight since unchanged code elements usually cease core logic whose test requirement ought to be preserved. If a mutant does not occur in the next version, being a brittle mutant, we consider it to stop standing, e.g., $M_{2,4}$ does not exist as $M_{3,4}$ due to deletion changes. From the example, we can observe when a system reaches maturity, as in the case of commons-io, the majority of the changes do not touch the core logic and most mutants $M_{1,n}$ are long-standing (still exist as $M_{3,n}$) precisely six out of eight. In particular, among various opportunities, this novel category suggests the potential reuse of past subsumption knowledge, avoids redundancy, addresses technical testing debt and aspires towards test completeness of mature code components.

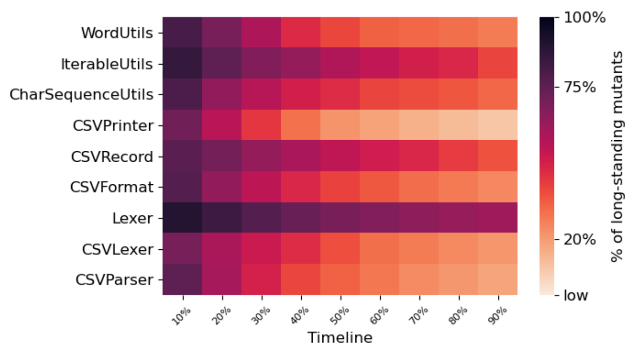


Figure 2: Mutant sets for observed files through studied history (timeline). Each cell in the heat map represents a normalised proportion of a subject history length, and the colour represents a percentage of mutants from the initial set over time through versions. The results show that mutants have diverse longevity, with brittleness, on average, of 52%.

3.2 Implementation Details - Mapping

To reuse mutants and follow their standing (w.r.t. how long a mutant 'stands' in one location without being altered), in this exploratory preliminary study, we map mutants considering changed lines and the context of change from git diff tool [9]. We take two program versions and map their code statements. Figure 1 indicates information of line numbers shift from version to version. *Note that for the purpose of this study, the history length of a mutant is computed from the first studied version till the last chronologically observed version.*

We start our analysis by extracting files with the most extended change history from the open-source mature Apache Commons projects. Then, we use the state-of-the-art PIT mutation testing tool [7] to generate mutants per each changed (committed) file, followed by the execution of tests and generation of the killing matrix. Next to the killing matrix, for each file, we keep metadata (info. about hunks, timestamps, mutants bytecode index, location etc.). Using extracted metadata, we create a regression history for each file, making a file-specific historical timeline. In the timeline of each file, a time-point represents a commit that introduces changes to the file. For each time-point, we calculate subsuming mutants (*reminder:* the mutants when distinguished, distinguish all others).

Besides the set of mutants, each time-point contains information about mapping changes to the consecutive points. Hence, long-standing mutation metric is a function $F(M_t, change\_map) = M'_t$. Where M_t is a mutant from time t, and $change\_map$ is a map containing information about code transition from time $t \longrightarrow t'$, while M'_t is the mutant at time t'. Formally, long-standing mutants definition is:

Definition. *Given n and m as timepoints of the first and last versions under the study of program P, where n > m. A mutant M is said to be long-standing if it exists on the same code element E throughout consecutive versions P^n, P^{n+1}, ... ,P^m until the point when the mutant M due to a committed program change does not appear on the code element E of a program version P^{m+1}.*

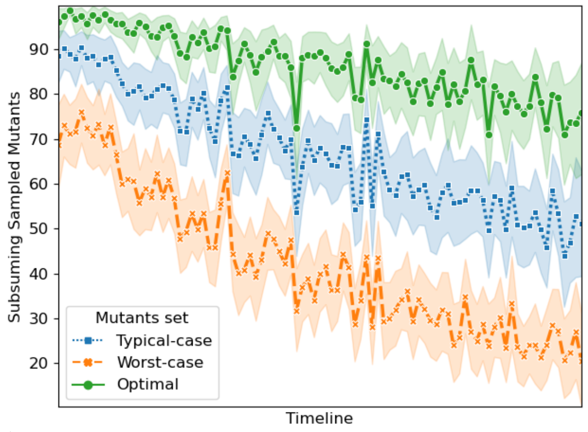
Given that mutants can 'stand' for several versions or just a few, we argue that the rate at which a mutation suite and mutant relevance decays over a series of releases suggests a mutation test brittleness. Knowing the degree to which mutants hold high-quality tests for a series of releases helps provide more prolonged test effectiveness. Accordingly, we introduce the mutation brittleness metric that measures mutants' longevity and assesses mutation suite test-effectiveness between series of versions.

4 INITIAL EVALUATION AND EARLY RESULTS

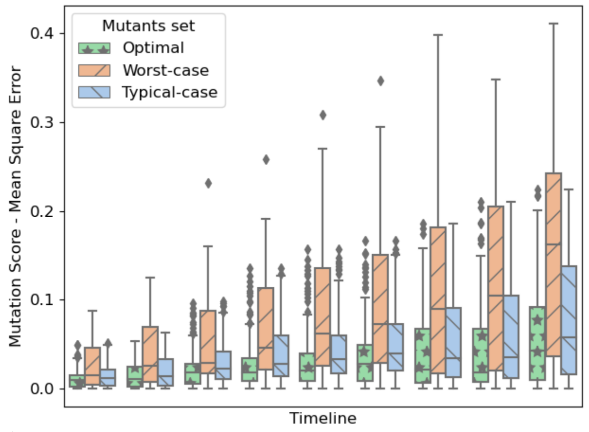
Evaluation data. Table 1 shows our subject data. From 4 different well maintained Apache Commons projects, we extracted nine files with the longest history of change and their corresponding commits. It is important to emphasize that due to technical reasons (e.g., PIT mutation testing tool requires green test suite to run), the time gap between commits is rather "longer" than one commit. Nevertheless, this did not stop us from mapping and observing long-standing mutants through the history of evolving systems, as strictly sequential commits can only improve our results.

Mutation Brittleness. Figure 2 depicts brittleness of mutants over time. In particular, it tells us how many mutants of a specific file exist through time, from their inception, on the same initial code elements, w.r.t., a code change has not altered mutants. From the figure, we can observe the diversity of the longevity distribution. Moreover, for the file Lexer, the ratio of long-standing mutants is significant, over 80% over observed time-points. On the contrary, we can see that the CsvPrinter file contains significant changes, and the ratio of the mutants degrade below 50% after the first half of observed points and below 20% in the second half of the observed history points. For other observed files, we see changes do not impact over 70% of mutants in the first quartile of the timeline and between 40-60 % for the rest of the time-points. These results demonstrate that mutants have a diverse lifetime over different evolution timelines, which suggests further investigation of whether mutants keep subsuming dynamic relationships over time and how mutant selection can affect test assessment.

Long-Standing Subsuming Mutants. Figure 3 demonstrates to what extent subsuming mutants convey their dynamic behaviour and how mutation selection can affect the test assessment capability of mutation testing over time. In particular, the figure depicts the scenario in which we select subsuming mutants at a certain point in time and observe the capacity in which they exist over time together with how well they can perform test assessment w.r.t., measuring mutation score. We randomly sample 10-30% of mutants (100 times to remove the threat of randomness; we choose these selection intervals as obviously selecting all mutants leads to traditional mutation testing) from each observed file and consider the file history length - *the figures show aggregated results since each subject has different history length.* Figure 3a illustrates the need for intelligent mutant selection as we can significantly distinguish between two sets, a) sets of mutants more optimal as they stand longer throughout observed history; hence longer enjoying mutant suites relevance and b) other sub-optimal sets that suffer relevance degradation, w.r.t., represent obsolete test requirements. Interestingly, optimal mutant selection promises continuous tracking of test effectiveness as the margin of degradation is ≈10% on the ratio of selected mutants. In comparison, we observe a worst-case sets of mutants which indicate obsolete test requirements and typical arbitrary sets as if there was no other way to select, then we would



(a) Long-Standing Subsuming mutants for different observed files throughout their studied *history*. An optimal set of mutants - when selected - shows a long-standing prospect. In contrast, a worst-case set of mutants when selected shows brittleness - indicating obsolete test requirements. A 'special relationship' between long-standing and subsuming mutants exists and indicates that subsuming mutants in one version are probably subsuming in the following versions.



(b) Mean Square Error of Mutation Score of initially selected and long-standing subsuming mutants. The optimal set of long-standing subsuming mutants demonstrates a capability to perform test assessments over time - preserving subsumption relationships - unlike the worst-case or typical sets which show higher MSE. An optimal set of long-standing mutant suites enjoy an order of magnitude longer relevance.

Figure 3

end up with a random. To observe how capable those mutants are of affecting test assessment over time - keeping their subsumption relationships - we calculate the mean square error (MSE) of mutation score (MS) between the initially selected set and those long-standing mutant sets. In Figure 3b we assess the difference in the MSE of MS between the optimal and suboptimal sets of long-standing mutants. We observe that the optimal set of long-standing subsuming mutants keeps MS high over time (low MSE $\approx$ 0.01%-0.04%), indicating a gradual loss in MS as the mutants stand longer in time, thus preserving mutant suite relevance. Accordingly, it is important to realize the potential in conveying knowledge of previously calculated dynamic relationships of mutants for at least 10x longer than a random selection. In particular, by selecting the sub-optimal sets of mutants, the threat of not preserving the knowledge appears, w.r.t., mutants less capable of test assessment over time, suggesting their low priority (higher MSE $\approx$0.01%-0.20%).

5 VISIONS OF THE FUTURE

We believe that long-standing mutants are an interesting category in their own right, worthy of further research and will serve to inspire the practitioners. They have implications not only for mutation testing, but also beyond mutation testing. In this section we set out future plans for further evaluation and investigation of the properties of long-standing mutants and their applications.

Implications regarding subsuming long-standing mutants: Despite showing that subsuming relationships can be preserved from version to version and that mutants' utility can be reused, we do not yet fully understand *why* subsuming mutants tend to last longer than subsumed mutants. A detailed study is needed to fully understand the subsumption and longevity drivers.

Implications for mutation testing tools: Our results also have implications for the development of future mutation testing tools. In particular, our results suggest the development of a robust mutant versioning system. Existing tools [5, 7, 21] focus on the generation of mutants, but not sophisticated mutant versioning. In future work, we need to investigate mutation testing tools that allow logging mutants' maturity, execution history, and fluctuation over time, supporting approaches that learn mutant behaviour and relating this to code changes. Previous work on flaky mutant detection [30], predictive modelling [1] and hyper-heuristics [11] (in particular that focused on mutation testing [32]) may form a good starting point for this research agenda.

Maximising long-standing mutant fault revelation: By focusing on long-standing mutants, we favour mutants that reside in relatively unchanging parts of the code. There is a natural concern that this may, in turn, lead to us favouring test suites that do not tend to reveal faults in changing parts of the code. Fortunately, the fact that a mutant lies in code region A does not render it insensitive to bugs that lie in (lexically separate) code region B. If there are transitive *dependencies* between A on B then we can expect high degrees of mutant coupling and even subsumption between the two regions. Suggesting future work to identify mutants that have high 'transitive dependence reach' through their transitive dependencies using techniques such as slicing [3] and chopping [12].

Implications of long-standing mutants beyond mutation testing research: The findings reported in this paper have implications beyond mutation testing to automated program repair[10, 20] and genetic improvement [18, 29]. It is often been argued that program repair is the inverse of mutation testing. Instead of inserting faults, repair seeks to remove them. Long-standing mutants are therefore also likely to find applications and implications in the field of program repair and genetic improvement research. For example, it would be interesting to explore 'long-standing' repairs as a counterpoint to long-standing mutants. One might reasonably conjecture that such repairs would remain relevant for longer than repairs in areas of code subject to high degrees of churn. However, the empirical assessment remains an open problem for future work.

Acknowledgment: This work is supported by the Luxembourg National Research Funds (FNR) through the CORE project grant C20/IS/14761415/TestFlakes.

REFERENCES

[1] Wasif Afzal and Richard Torkar. 2011. On the application of genetic programming for software engineering predictive modeling: A systematic review. *Expert Systems Applications* 38, 9 (2011), 11984–11997.

[2] Paul Ammann, Marcio Eduardo Delamaro, and Jeff Offutt. 2014. Establishing theoretical minimal sets of mutants. In *2014 IEEE seventh international conference on software testing, verification and validation*. IEEE, 21–30.

[3] David Binkley, Nicolas Gold, Mark Harman, Syed Islam, Jens Krinke, and Shin Yoo. 2014. ORBS: Language-Independent Program Slicing. In 22nd *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2014)*. Hong Kong, China, 109–120.

[4] Mark Anthony Cachia, Mark Micallef, and Christian Colombo. 2013. Towards incremental mutation testing. *Electronic Notes in Theoretical Computer Science* 294 (2013), 2–11.

[5] Thierry Titcheu Chekam, Mike Papadakis, and Yves Le Traon. 2019. Mart: a mutant generation tool for LLVM. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*. ACM, 1080–1084. https://doi.org/10.1145/3338906.3341180

[6] Thierry Titcheu Chekam, Mike Papadakis, Yves Le Traon, and Mark Harman. 2017. An Empirical Study on Mutation, Statement and Branch Coverage Fault Revelation that Avoids the Unreliable Clean Program Assumption. *IEEE/ACM International Conference on Software Engineering*.

[7] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. 2016. PIT: a practical mutation testing tool for Java (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*. ACM, 449–452. https://doi.org/10.1145/2931037.2948707

[8] Aayush Garg, Milos Ojdanic, Renzo Degiovanni, Thierry Titcheu Chekam, Mike Papadakis, and Yves Le Traon. 2022. Cerebro: Static Subsuming Mutant Selection. *IEEE Transactions on Software Engineering* (2022), 1–1. https://doi.org/10.1109/TSE.2022.3140510

[9] Git. 2022. Git-Diff. Retrieved September 29, 2022 from https://git-scm.com/docs/git-diff

[10] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (2019), 56–65.

[11] Mark Harman, Edmund Burke, John A. Clark, and Xin Yao. 2012. Dynamic Adaptive Search Based Software Engineering (Keynote Paper). In 6th *IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM 2012)*. Lund, Sweden, 1–8.

[12] Daniel Jackson and Eugene J. Rollins. 1994. A New Model of Program Dependences for Reverse Engineering. In *Symposium on the Foundations of Software Engineering (FSE '94)*. 2–10.

[13] Yue Jia and Mark Harman. 2009. Higher order mutation testing. *Information and Software Technology* 51, 10 (2009), 1379–1393.

[14] Yue Jia and Mark Harman. 2009. Higher Order Mutation Testing. *Journal of Information and Software Technology* 51, 10 (2009), 1379–1393.

[15] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (September–October 2011), 649 – 678.

[16] Marinos Kintis, Mike Papadakis, and Nicos Malevris. 2010. Evaluating Mutation Testing Alternatives: A Collateral Experiment. In *17th Asia Pacific Software Engineering Conference, APSEC 2010, Sydney, Australia, November 30 - December 3, 2010*. IEEE Computer Society, 300–309. https://doi.org/10.1109/APSEC.2010.42

[17] Bob Kurtz, Paul Ammann, Jeff Offutt, and Mariet Kurtz. 2016. Are We There Yet? How Redundant and Equivalent Mutants Affect Determination of Test Completeness. *IEEE International Conference on Software Testing, Verification and Validation*, 142–151. https://doi.org/10.1109/ICSTW.2016.41

[18] William B. Langdon and Mark Harman. 2010. Evolving a CUDA Kernel from an nVidia Template. In *2010 IEEE World Congress on Computational Intelligence*, Pilar Sobrevilla (Ed.). IEEE, Barcelona, 2376–2383. https://doi.org/doi:10.1109/CEC.2010.5585922

[19] Wei Ma, Thierry Titcheu Chekam, Mike Papadakis, and Mark Harman. 2021. MuDelta: Delta-Oriented Mutation Testing at Commit Time. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 897–909. https://doi.org/10.1109/ICSE43902.2021.00086

[20] Alexandru Marginean, Johannes Bader, Satish Chandra, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, and Andrew Scott. 2019. SapFix: Automated End-to-End Repair at Scale. In *International Conference on Software Engineering (ICSE) Software Engineering in Practice (SEIP) track*. Montreal, Canada.

[21] Kevin Moran, Michele Tufano, Carlos Bernal-Cárdenas, Mario Linares-Vásquez, Gabriele Bavota, Christopher Vendome, Massimiliano Di Penta, and Denys Poshyvanyk. 2018. Mdroid+: A mutation testing framework for Android. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. IEEE, 33–36.

[22] A.J. Offutt, G. Rothermel, and C. Zapf. 1993. An experimental evaluation of selective mutation. In *Proceedings of 1993 15th International Conference on Software Engineering*. 100–107. https://doi.org/10.1109/ICSE.1993.346062

[23] Milos Ojdanic, Wei Ma, Thomas Laurent, Thierry Titcheu Chekam, Anthony Ventresque, and Mike Papadakis. 2022. On the use of commit-relevant mutants. *Empir. Softw. Eng.* 27, 5, 114. https://doi.org/10.1007/s10664-022-10138-1

[24] Milos Ojdanic, Ezekiel Soremekun, Renzo Degiovanni, Mike Papadakis, and Yves Le Traon. 2022. Mutation Testing in Evolving Systems: Studying the Relevance of Mutants to Code Evolution. *ACM Trans. Softw. Eng. Methodol.* (apr 2022). https://doi.org/10.1145/3530786 Just Accepted.

[25] Elmahdi Omar, Sudipto Ghosh, and Darrell Whitley. 2013. Constructing subtle higher order mutants for Javer and AspectJ programs. In *International Symposium on Software Reliability Engineering (ISSRE'13)*. IEEE, 340–349.

[26] Mike Papadakis, Thierry Titcheu Chekam, and Yves Le Traon. 2018. Mutant Quality Indicators. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops, Västerås, Sweden, April 9-13, 2018*. IEEE Computer Society, 32–39. https://doi.org/10.1109/ICSTW.2018.00025

[27] Mike Papadakis, Yue Jia, Mark Harman, and Yves Le Traon. 2015. Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple, Fast and Effective Equivalent Mutant Detection Technique. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*. IEEE Computer Society, 936–946. https://doi.org/10.1109/ICSE.2015.103

[28] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Chapter Six - Mutation Testing Advances: An Analysis and Survey. Advances in Computers, Vol. 112. Elsevier, 275–378. https://doi.org/10.1016/bs.adcom.2018.03.015

[29] Justyna Petke, Saemundur O. Haraldsson, Mark Harman, William B. Langdon, David R. White, and John R. Woodward. 2018. Genetic Improvement of Software: a Comprehensive Survey. *IEEE Transactions on Evolutionary Computation* 22, 3 (June 2018), 415–432. https://doi.org/doi:10.1109/TEVC.2017.2693219

[30] August Shi, Jonathan Bell, and Darko Marinov. 2019. Mitigating the effects of flaky tests on mutation testing. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, Dongmei Zhang and Anders Møller (Eds.). ACM, 112–122. https://doi.org/10.1145/3293882.3330568

[31] Xiangjuan Yao, Mark Harman, and Yue Jia. 2014. A Study of Equivalent and Stubborn Mutation Operators Using Human Analysis of Equivalence. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(IEEE/ACM International Conference on Software Engineering 2014)*. Association for Computing Machinery, New York, NY, USA, 919–930. https://doi.org/10.1145/2568225.2568265

[32] Jie Zhang, Lingming Zhang, Mark Harman, Dan Hao, Yue Jia, and Lu Zhang. 2019. Predictive Mutation Testing. *IEEE Transactions on Software Engineering* 45, 9 (2019), 898–918. https://doi.org/10.1109/TSE.2018.2809496

[33] Lingming Zhang and Darko Marinov. 2012. Regression Mutation Testing. *The ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 341.

Received 2023-05-03; accepted 2023-07-19