Mutation Operators for Mutation Testing of Angular Web Applications

Sarah Augustin, Hendrik Winkelmann[a] and Herbert Kuchen[b]
Department of Information Systems, University of Münster, Leonardo-Campus 3, Münster, Germany

Keywords: Mutation Testing, Mutation Operators, Angular, Web Applications.

Abstract: Mutation testing is an approach for assessing the quality of a test suite by using mutation operators to insert changes into the code and then checking whether the test suite can detect the inserted changes. Due to the growing prevalence and complexity of web applications, the importance of web testing has increased, making mutation testing a potentially beneficial approach for web applications. Since in web applications, mostly web-specific mistakes and not generic mistakes occur, the question arises, to whether new mutation operators simulating such realistic, web-specific mistakes perform better than the traditional, generic mutation operators. The work at hand addresses this question by developing new mutation operators specific to the client-side TypeScript code of Angular web applications and evaluating how they perform in comparison to the traditional mutation operators. The findings indicate that the new web-specific mutation operators introduce fewer, more realistic, and harder-to-kill mutants than the traditional mutation operators, thus being a promising approach for assessing the test suite quality of web applications.

1 INTRODUCTION

Since the 1970s, mutation testing has gained increasing attention from both academia and industry, leading to, e.g., its integration into Google's mandatory code review process (Petrović and Ivanković, 2018; Jia and Harman, 2011). Mutation testing can be applied to a variety of different types of applications including web applications. Applying mutation testing to web applications can contribute to ensuring a high quality of web tests. Since, according to Marchetto et al., almost three-quarters of the bugs in web applications are web-specific, the question arises whether mutation testing can be applied to web applications with its traditional mutation operators or whether new mutation operators specifically targeting such web-specific faults would perform better (Marchetto et al., 2009).

The work at hand addresses this question by developing new mutation operators inserting web-specific faults into the application code during mutation testing. More specifically, eight new mutation operators for mutating code related to comparisons, error handling, input default values, input validation, input names, subscribe calls, unsubscribe calls, and RxJS operators were introduced. Those new mutation operators are compared to the traditional mutation operators in order to assess whether inserting realistic, web-specific faults provides benefits compared to inserting the traditional, generic faults. Since it can be argued that faults depend on the web framework used, this work specifically considers web applications built with the Angular framework. Angular was chosen because it is one of the most popular web frameworks (Malcher et al., 2023). Following the finding of Ocariza et al. that the majority of faults arise neither in the server-side code nor in the HTML code, but instead in the client-side code, this work specifically focuses on the client-side TypeScript code of Angular web applications (Ocariza et al., 2013, p. 56).

This work contributes to theory by providing proof by construction that creating mutation operators simulating realistic mistakes is a promising approach worthy of being researched in depth. The contribution towards practice corresponds to providing software developers with new mutation operators for mutation testing of Angular web applications. If used instead of the traditional mutation operators, the new mutation operators contribute to speeding up the process of mutation testing and reducing the human effort needed to check the output of mutation testing.

This work is organized as follows. While Section 2 describes the research method, an overview of the theoretical background is provided in Section 3. In turn, Section 4 the design of the new mutation operators is illustrated. The implementation and evaluation are presented in Section 5 and Section 6. The work finishes with a conclusion in Section 7.

[a] https://orcid.org/0000-0002-7208-7411
[b] https://orcid.org/0000-0002-6057-3551

Augustin, S., Winkelmann, H. and Kuchen, H.
Mutation Operators for Mutation Testing of Angular Web Applications.
DOI: 10.5220/0013203900003928
In *Proceedings of the 20th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2025)*, pages 390-397
ISBN: 978-989-758-742-9; ISSN: 2184-4895

2 RESEARCH METHOD

For the design of new mutation operators simulating realistic programming mistakes within the TypeScript files of Angular web applications, first, those realistic mistakes need to be identified. To identify the mistakes, two research methods were chosen: a systematic literature review and semi-structured expert interviews.

The systematic literature review was conducted according to the work of vom Brocke et al. and included searching the database Scopus with a search string and extending the results with a forward and backward search (vom Brocke et al., 2009). To enrich the results of the literature review, additionally, four semi-structured expert interviews were conducted. All four interviewees are software developers working full-time in the IT industry and have developed Angular web applications professionally. One of the interviewees also gives professional training courses on the topic of Angular web applications. The programming mistakes identified through the systematic literature review and the interviews were grouped into categories.

To narrow down the scope, some categories were selected to be focused upon for the design of the new mutation operators. The categories were chosen based on the criteria that they should be applicable to all Angular web applications, that they should have been included in at least two sources, and that at least one of the two sources had to be an interview. These criteria aim at selecting categories of high relevance by ensuring that they have occurred in several sources. The decision to require that the category must have at least come up in one interview was made because the interviews, in contrast to the papers, were specific to Angular rather than general for all web applications. For each of the mistakes in the remaining categories, a new mutation operator was designed that simulates it.

3 THEORETICAL BACKGROUND AND RELATED WORK

First, we explain mutation testing in detail in Subsection 3.1. Afterward, in Subsection 3.2, more details about advances and research in the field of mutation testing are presented.

3.1 Traditional Mutation Testing

Mutation testing is an approach used to assess the quality of a test suite. When applying mutation testing to a program, mutants are automatically created for that program (Offutt and Untch, 2001). A mutant is a version of the original program in which a small change was made (Offutt and Untch, 2001). Those small changes are inserted by applying transformation rules called mutation operators (Mike Papadakis et al., 2019; Jia and Harman, 2011). The test suite can be run on each mutant to check whether at least one test fails. If so, the mutant is called dead or killed, otherwise, it is called alive or a surviving mutant (Offutt and Untch, 2001). Equivalent mutants which correspond to functionally equivalent versions of the original program are removed from the group of the surviving mutants (Mike Papadakis et al., 2019). The remaining surviving mutants point out inadequacies of the test suite insofar as the inserted defect was not detected (Offutt and Untch, 2001). The test suite can be improved by adding new test cases killing these mutants (Offutt and Untch, 2001).

3.2 Advances and Research in Mutation Testing

Research in the field of mutation testing has covered various aspects. One research approach aiming at inserting more realistic and complex faults is the "higher order mutation testing" in which mutation operators are applied several times to create a mutant (Harman et al., 2010). The disadvantage of higher-order mutation testing is the immense number of possible fault combinations, which can make it too computationally expensive and impractical (Jia and Harman, 2009; Harman et al., 2010). The approach of higher-order mutation testing is similar to the approach of this work insofar as both aim at inserting more realistic faults. This work circumvents the disadvantage of the search space explosion by using domain knowledge. Instead of all combinations of first-order faults, only realistic faults are considered.

Another research field is the reduction of the computational cost of mutation testing as this is one of its main problems (Jia and Harman, 2011). According to Offutt and Untch, research on reducing those computational expenses can typically be categorized into one of the three strategies of "do fewer, do smarter, or do faster" (Offutt and Untch, 2001, p. 37). The idea of the "do fewer" approach is to reduce the number of mutants without an intolerable information loss (Offutt and Untch, 2001). One approach in literature for reducing the number of mutants is, for example, selective mutation, with which some mutation operators are selected according to a certain criterion and only those mutation operators are used to generate mutants (Mike Papadakis et al., 2019; Jia and Har-

man, 2011). The category of "do fewer" strategies can also contribute to mitigating human oracle problem which corresponds to the high amount of human effort needed to check the output of the mutation testing (Jia and Harman, 2011). If, with the "do fewer" approach, fewer mutants are generated, it can be argued that checking the output of the mutation testing takes less effort. This work could be classified as a "do fewer" approach if the new mutation operators are used instead of the existing mutation operators, as they generate fewer mutants.

Research has also addressed the development of new mutation operators for various scenarios. Some researchers have introduced mutation operators for inserting a specific type of fault into the code (Mike Papadakis et al., 2019). To the best of the authors' knowledge, however, no research has yet focused on developing a set of web-specific mutation operators for client-side TypeScript code.

4 DESIGN OF THE MUTATION OPERATORS

Following the research method described in Section 2, a systematic literature review and four semi-structured expert interviews were conducted. The literature review identified 39 different programming mistakes grouped into 15 categories, while the expert interviews identified 20 different programming mistakes grouped into 11 categories. When combining those mistakes, a total of 55 mistakes in 23 categories was identified. The number does not correspond to the sum since some mistakes were identified in both the literature review and the interviews. Applying the criteria described in Section 2, four categories were selected to focus upon: comparisons, error handling, forms, and RxJS. RxJS is a library for reactive programming commonly used in Angular (Malcher et al., 2023). Those four categories include a total of eight mistakes. For each mistake, a new mutation operator was designed that simulates it. To the best of the authors' knowledge, the new mutation operators are all novel, as no web-specific mutation operators for client-side TypeScript code operators have been introduced in literature yet. In the following subsections, the new mutation operators are described in detail.

4.1 Comparison Mutation Operator

The first new mutation operator called "comparison mutation operator" simulates the programming mistake of using a loose comparison operator instead of a strict one or vice versa. The loose operators "==" and "!=" perform first an automatic type conversion if the left- and right-hand sides are of a different data type and then compare the values for equality (Mozilla, 2023). The strict operators "===" and "!==" in contrast do not perform a type conversion (Mozilla, 2023). The new comparison mutation operator simulates the mistake of confusing them by switching "==" to "===", "!=" to "!==", "===" to "==", and "!==" to "!=" in the TypeScript code. Test cases which depend on the comparisons always resulting in the expected boolean values can, for example, be used to kill such mutants.

4.2 Error Handling Mutation Operator

The second new mutation operator is called "error handling mutation operator". It simulates the mistake of not catching and handling an exception. In the TypeScript files of Angular web applications, errors can occur both synchronously and asynchronously. Asynchrony in Angular is typically handled by either the datatype `Observable` or the datatype `Promise`. Depending on where the error occurs, error handling can take place in four different ways:

1. catching synchronous errors in a try-catch block
2. catching asynchronous errors in a `Promise`
3. catching asynchronous errors in the `pipe` method of an `Observable`
4. catching asynchronous errors in the `subscribe` block of an `Observable`

The new error handling mutation operator deletes the code responsible for catching and handling such errors. The mutants generated by this mutation operator can, for example, be killed by test cases which purposefully trigger those errors and test whether the application handles those errors properly.

4.3 Input Default Value Mutation Operator

The third new mutation operator is called "input default value mutation operator". It simulates the mistake of setting an incorrect default value for a form field. This is done by mutating the default values of form fields. If the default value is an empty string, it is replaced by 'mutated string'. If it is a non-empty string it is replaced by an empty string. If the default value is a numeric literal, its value is increased by one, and if it is a boolean literal, it is inverted. In Angular, the most common type of form is the "reactive form" (Malcher et al., 2023). A reactive form, including its

default values, can be defined in various ways. The input default value mutation operator supports all those various ways. The mutants generated by this mutation operator can, for example, be killed by test cases using the default value of a form field instead of an individually entered value.

4.4 Input Validation Mutation Operator

The fourth new mutation operator is called "input validation mutation operator". It simulates the mistake of forgetting to set up input validation for a form field or an entire form. The mutation operator thus deletes the client-side input validation which ensures that the user has entered valid data into the form before submitting it. In general, several ways of defining input validation for reactive forms exist in Angular. Independent of which way is used, the new input validation mutation operator deletes the input validation. The mutants generated by this mutation operator can, for example, be killed by test cases purposefully inserting invalid data into a form and checking whether this case is handled properly. It thus encourages writing test cases representing unexpected user behavior.

4.5 Input Name Mutation Operator

The fifth new mutation operator is called "input name mutation operator". It simulates the mistake of assigning an incorrect name to a form field. The input name mutation operator thus changes the name of a form field by appending the postfix "_mutated" to it. As explained in further detail in Section 5, due to technical reasons, it was not possible to mutate names that were defined as class properties. All other ways of how names can be set are supported by the new mutation operator. This mutation operator encourages writing test cases checking whether all input fields work as expected and process their values correctly, which would not be possible if an incorrect name would be assigned.

4.6 Subscribe Call Mutation Operator

The sixth new mutation operator, which is called "subscribe call mutation operator", simulates the mistake of forgetting to subscribe to an `Observable`. Since an `Observable` is lazy and is only invoked if it is being subscribed to, forgetting the subscription has the effect that the `Observable` is never executed (Lesh et al., 2022). In Angular, an `Observable` is subscribed by calling the `subscribe` method on it. The new mutation operator deletes the subscribe call including everything that is passed to the method. Mutants generated by this mutation operator can be killed by adding test cases which depend on the `Observable` to actually be invoked.

4.7 Unsubscribe Call Mutation Operator

The seventh new mutation operator is called "unsubscribe call mutation operator". It simulates the mistake of forgetting to unsubscribe an `Observable`. Unsubscribing is necessary if the `Observable` never completes on its own and if its execution is thus infinite (Lesh et al., 2022). Without unsubscribing, computation power and memory resources are wasted, potentially causing memory leaks (Lesh et al., 2022).

Two ways of unsubscribing from an `Observable` exist. First, the `unsubscribe()` method can be called on an existing subscription (Malcher et al., 2023). Second, when creating an `Observable`, the `pipe()` method can be called on it. To the `pipe` method, several RxJS operators can be passed which manipulate the emitted stream of values. The five RxJS operators `first()`, `take()`, `takeUntil()`, `takeWhile()`, and `takeUntilDestroyed()` can be used to automatically unsubscribe from the `Observable`, e.g., based on a certain condition passed to the operator (Malcher et al., 2023). The new mutation operator deletes both the unsubscribe calls and the five listed RxJS operators from the TypeScript files of the Angular web application. This mutator encourages writing test cases which depend on the `Observable` to be finished as required.

4.8 RxJS Mutation Operator

The eighth and last new mutation operator is the "RxJS mutation operator". It simulates the mistake of using the wrong RxJS operator. According to the official RxJS website, over one hundred operators such as `map()`, `reduce()`, and `switchMap()` exist, each of which can be applied to the values emitted by an `Observable` (Lesh et al., 2023). The new RxJS mutation operator switches the RxJS operators within the TypeScript code to another one. As explained in further detail in Section 5, due to technical reasons, it is not possible to switch an RxJS operator to an arbitrary other RxJS operator, but only to another RxJS operator which has already been imported into the file. The RxJS operators are thus switched with the first imported operator that has a different name than itself. The mutants generated by this mutation operator can be killed by test cases which depend on the values of the Observable to be changed as required by the context.

5 IMPLEMENTATION OF THE MUTATION OPERATORS

For the implementation of the new mutation operators, the existing mutation testing tool StrykerJS was taken as a starting point and was extended by the new mutation operators.[1] StrykerJS was selected because it is the most common mutation testing tool for JavaScript, because it is an open-source project, and because it already offers sixteen built-in mutation operators to which the new mutation operators can be compared (Sánchez et al., 2022; Info Support, 2023).

StrykerJS performs the mutations not in the TypeScript code directly but in the abstract syntax tree (AST) of the TypeScript code. When making a mutation, StrykerJS replaces one node in the AST by another node. This new node can, e.g., be created completely new or can be created as a copy of the existing node and then be modified. The new AST node is then transformed back into TypeScript code by StrykerJS.

Due to performance reasons, StrykerJS uses an approach called mutation switching for conducting mutation testing (Jansen, 2020). With mutation switching, all mutants are inserted into the same file (Jansen, 2020). By using, e.g., if-else-constructs, the mutants can be activated and deactivated in that file, so that for each run of the test suite only one mutant is active (Jansen, 2020). This has a performance advantage since only this one file needs to be transpiled and compiled (Jansen, 2020). One disadvantage, however, is that it allows only mutations of AST nodes, which are located at places where if-else-constructs are allowed. Due to this reason, the RxJS mutation operator cannot switch an RxJS operator to an non-imported other RxJS operator, since the import statement cannot be changed. Similarly, this is also the reason for why the input name mutation operator cannot mutate names that are defined as class properties.

6 EVALUATION OF THE MUTATION OPERATORS

For the evaluation, the new mutation operators were applied to an existing Angular web application. This web application is a desk sharing tool developed commercially. For unit testing of the web application, the testing framework Jest is used. Currently, 24 test suites with a total of 138 tests exist.

Mutation testing was applied with three different sets of mutation operators to this web application: with only the eight new mutation operators, with only the 16 mutation operators which were already pre-implemented in StrykerJS, and with all 24 mutation operators in combination. In the following subsections, first, the result of the mutation testing is presented. Then the execution time, the realism of the mistakes, and expert feedback are discussed in further detail. Afterward, the findings are combined for a final discussion.

6.1 Mutation Testing Results

For all three versions – for applying only the new, only the traditional, or all mutation operators – the distribution of the mutants on the four different statuses killed, survived, no coverage, and timeout was determined. No coverage means that the mutant survived because it was not even covered by any test. Timeout means that StrykerJS has stopped the execution due to it taking unexpectedly long. This can, for example, happen if accidentally an infinite loop is created by the change made by the mutation operator. The two further statuses of compile errors and runtime errors are not considered, since they did not occur. The distribution in percent can be seen in Table 1. In absolute numbers, the eight new mutation operators generated a total number of 192 mutants, while the 16 existing, traditional mutation operators generated a total 1114 mutants.

The mutation testing result of the eight new mutation operators can be beneficial in several ways. First of all, the results can be used to improve the quality of the test suites. For example, for the 58,85% of mutants that survived, it could be checked which tests have covered them and why they did not detect the inserted mistake. Those tests can then be improved so that they afterwards catch the inserted mistakes, thus ensuring that if such mistakes are accidentally inserted in the future, they will be caught. Similarly, the 21,35% of mutants that were not covered can be used to get insights into which parts of the code are not covered by tests yet. This information can motivate the developer to write additional tests covering that part of the code. The mutation testing result might additionally contribute to finding mistakes by directing the programmer's attention to the critical parts of

Table 1: Status distribution of the mutants per mutation operator type in percent.

Mutation operators	Killed	Survived	No coverage	Timeout
New	19,79%	58,85%	21,35%	0,00%
Traditional	29,17%	33,30%	37,16%	0,36%
All	27,79%	37,06%	34,84%	0,31%

[1] https://stryker-mutator.io/docs/stryker-js/introduction/

the code in which common mistakes occur.

Comparing the mutation testing results of the new mutation operators to the existing, traditional mutation operators, one can see that 19.79% of the mutants generated by the new mutation operators were killed, while in contrast 29.17% of the mutants generated by the traditional mutation operators were killed. That indicates that the new mutation operators generate harder-to-kill mutants, which therefore potentially provide more value and insights to the programmer.

Looking at the column entitled "survived", one can see that 58.85% of the mutants generated by new mutation operators survived, while only 33.30% of the traditional mutation operators survived. This shows that the new mutation operators generate a higher percentage of mutants that are covered by a test but that are still not killed. This might indicate that the new mutation operators have a higher potential for improving the existing test cases that cover those survived mutants. The programmer can check which test case covered the mutant and can improve it to catch the mistake inserted by the mutation operator.

In the next column, one can see that 21.35% of the mutants generated by the new mutation operators were not covered while 37.16% of the mutants generated by the traditional mutation operators were not covered. This might indicate two things. First, the new mutation operators seem to generate more mutants in areas of code already covered by tests than the traditional mutation operators, confirming that the tests address relevant code areas which often include mistakes. Second, the traditional mutation operators seem to perform better at highlighting uncovered parts of the code than the new mutation operators. Towards that goal, however, other metrics such as code coverage can be used instead of mutation testing.

All in all, it becomes clear that the traditional mutation operators highlight more areas of the code that are not covered yet, thus motivating the programmer to write more tests for the uncovered parts of the code. At the same time, the new mutation operators insert more changes at locations already covered by test cases than the traditional mutation operators but still achieve a lower percentage of killed mutants. This shows that they insert hard-to-kill changes that require the existing test cases to be improved.

6.2 Execution Time

Mutation testing was executed on the web application in three versions: with only the eight new mutation operators, with only the 16 traditional mutation operators, and with all 24 mutation operators. As was already mentioned in the previous subsection, the new mutation operators inserted 192 mutants. This makes an average of 24 mutants per new mutation operator. The traditional mutation operators inserted 1114 mutants, corresponding to an average of around 69 mutants per traditional mutation operator. Since the number of mutants is lower for the new mutation operators, this indicates that using the new mutation operators instead of the traditional mutation operators might be beneficial in terms of execution time.

Table 2: The execution time of mutation testing.

Mutation operators	Average execution time
New	21min
Traditional	92min
All	112min

The execution time was determined for the three versions of using only the new, only the traditional, and all mutation operators. Since the execution time is influenced by the timeout setting of StrykerJS, the timeout was set to 30 seconds across all runs. Since the execution time might vary per run, four runs were conducted, and the average was calculated. The results rounded to minutes can be seen in Table 2. As can be seen in the table, the new mutation operators have a lower execution time than the traditional mutation operators.

6.3 Past Mistakes

To evaluate the new mutation operators, it was checked whether the mistakes inserted by them have already occurred in the web application in the past. If they did, this indicates that the new mutation operators are of relevance insofar that they might have helped prevent those errors as they would have motivated improving the test suite in a way to catch such errors and would have generally raised awareness for these types of errors. To gain insights into the mistakes made in the past, both the bug tickets and the code review findings were consulted. In the following, for each new mutation operator, the bug tickets and code review findings related to that new mutation operator are briefly discussed.

The mistake of using a loose comparison operator instead of a strict one, which is simulated by the first new mutation operator, was uncovered in one code review. The mistake simulated by the second mutation operator, i.e., the mistake of forgetting to catch an error, was addessed in one code review finding and two bug tickets. The third mutation operator is the mutation operator artificially inserting the mistake of setting a wrong default value. This mistake was found in one bug ticket as well. The fourth mutation operator

simulates the mistake of forgetting input validation. This mistake was addressed in three bug tickets and one code review finding. The fifth and sixth new mutation operator are the mutation operators for changing the name of a form field and for forgetting to subscribe to an `Observable`. For those two mistakes, no occurrence could be found. This could be due to the reason that both of these errors might become apparent already when testing the implementation locally during development. Therefore, maybe, the mistakes are noticed too early to be found in a code review or bug ticket. The seventh mutation operator is the one simulating the mistake of forgetting to unsubscribe an `Observable`. This mistake was found in two code review findings. The eighth and last new mutation operator is the one simulating the wrong use of the RxJS operators. This mistake was also addessed in one code review.

For the traditional mutation operators in StrykerJS, it was also checked whether matching bug tickets or code review findings exist. For those mutation operators, however, only mistakes corresponding to a removal of the `sort()` method, an inverted boolean value, and several wrong string literals were found. It thus becomes apparent that the new mutation operators are, in comparison to the traditional mutation operators, more successful in simulating real-world programming mistakes.

6.4 Developer Feedback

To evaluate the new mutation operators, a semi-structured expert interview was conducted with the main developer of the web application under consideration. He set up the initial application, has been on the team ever since, and is the team member doing most code reviews.

Even though he had not come across mutation testing in practice yet, the developer said he would expect mutation testing to provide an additional level of quality. According to the developer, the mistakes occurring in Angular web applications are typically not generic mistakes, but mostly frontend-, web- and Angular-specific mistakes. This indicates that the traditional mutation operators inserting generic mistakes do not represent typical mistakes in Angular web applications. This was also supported by the answers of the developer after being shown the traditional mutation operators implemented into the StrykerJS tool. He said that most of those mistakes are unlikely to happen in reality, especially the very basic changes like switching an operator. The only mutation operator mentioned to maybe be realistic was the one changing the condition of, e.g., a loop or an if statement. Most of the traditional mutation operators, however, would not be realistic. Additionally, it was said that if mistakes such as the ones inserted by the traditional mutation operators were made, they would be recognized rather early. Thus, the interview not only suggests that the traditional mutation operators do not represent realistic mistakes, but also that if such mistakes occur, they are recognized early, therefore indicating a lower need for test cases catching those types of mistakes. The new mutation operators, in contrast, were assessed as mostly being very realistic by the developer.

The developer stated that he would prefer the new mutation operators over the traditional mutation operators, indicating that the new mutation operators pose a useful contribution to practice. He would not use both the new and the traditional mutation operators in combination, because of the human effort and limited time of the developers required for checking the mutation testing result and fixing the test suite. If, however, the human effort would be lower than expected, more mutation operators could be added. Overall, the interview indicates that the new mutation operators are an improvement over the traditional mutation operators and provide benefits for practitioners in the field of Angular web applications.

6.5 Discussion

Summarizing the results of the four previous subsections, it can be said that the eight new mutation operators successfully insert more realistic mistakes than the traditional mutation operators. The new mutation operators were additionally found to introduce on average fewer mutants into the code than the traditional mutation operators. If used instead of the traditional mutation operators, they can contribute to reducing the execution time and possibly also the human effort. Another insight gained is that – in the case of the web application under consideration – the new mutation operators inserted more changes into the areas of code already covered by tests, whereas the traditional mutation operators inserted more changes into the uncovered areas. The traditional mutation operators thus motivate the developers to write more test cases for uncovered parts of the code, while the new mutation operators motivate the developers more to improve the test cases which already cover the mutant but do not manage to kill it. An additional insight is that even though the new mutation operators inserted a higher percentage of mutants into areas of code already covered by tests, their use still resulted in a lower rate of killed mutants than the traditional mutation operators. This might indicate that the new

mutation operators insert hard-to-kill changes which require more specific test cases.

7 CONCLUSIONS

In this work, eight new mutation operators for the TypeScript files of Angular web applications were introduced. Towards that goal, typical mistakes made by programmers when developing Angular web applications were identified, and, based on them, new mutation operators were designed that simulate such mistakes. The mutation operators were implemented to allow applying them to Angular web applications in practice. Finally, the results were demonstrated and evaluated to allow assessing whether the new mutation operators provide additional benefits compared to the existing, traditional mutation operators.

The promising results of the evaluation show that the new mutation operators can contribute to practice by allowing Angular developers to better evaluate the quality of their test suites. Practitioners can apply the new mutation operators to their Angular web applications and can use the mutation testing results as guidelines to assess the current quality of the test suites and to improve it so that both current and future mistakes in the application code are more likely to be identified. When using the new mutation operators instead of the traditional mutation operators, they can additionally contribute to reducing the execution time and human effort associated with mutation testing.

Threats to validity consist of the evaluation of the mutation operators on a single web application and by consulting only the main developer. To confirm the generalizability of the findings, as part of future work, the new mutation operators should be tried out on more Angular applications and more stakeholders should be interviewed to gain their insights on the potential benefits and drawbacks. Similarly, a threat to validity is that the interviews used to identifying common mistakes might be biased based on personal experiences and having a similar background. As part of future work, more interviews with developers of various levels of experience might be conducted and more mutation operators could be added that simulate the newly identified common mistakes. Another possibility for future work would be to apply the approach of this work to other domains or frameworks. All in all, it becomes clear that the promising results of this work open up a lot of areas of future work, illustrating the high research potential of mutation operators simulating realistic mistakes in the field of mutation testing.

REFERENCES

Harman, M., Jia, Y., and Langdon, W. B. (2010). A manifesto for higher order mutation testing. In *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, pages 80–89. IEEE.

Info Support (2023). Stryker mutator: Supported mutators.

Jansen, N. (2020). Stryker mutator: Announcing stryker 4.0 - mutation switching.

Jia, Y. and Harman, M. (2009). Higher order mutation testing. *Information and Software Technology*, 51(10):1379–1393.

Jia, Y. and Harman, M. (2011). An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678.

Lesh, B., Driscoll, D., Wortmann, J.-N., Jamieson, N., Taylor, P., and Lee, T. (2022). Observable.

Lesh, B., Driscoll, D., Wortmann, J.-N., Jamieson, N., Taylor, P., and Lee, T. (2023). Rxjs operators.

Malcher, F., Koppenhagen, D., and Hoppe, J. (2023). *Angular: Das große Praxisbuch - Grundlagen, fortgeschrittene Themen und Best Practices*. ix Edition. dpunkt Verlag, Heidelberg, 4., überarbeitete und aktualisierte auflage edition.

Marchetto, A., Ricca, F., and Tonella, P. (2009). An empirical validation of a web fault taxonomy and its usage for web testing. *Journal of Web Engineering*, 8(4):316–345.

Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman (2019). Chapter six - mutation testing advances: An analysis and survey. In Atif M. Memon, editor, *Advances in Computers*, volume 112, pages 275–378. Elsevier.

Mozilla (2023). Mdn web docs: Equality comparisons and sameness.

Ocariza, F., Bajaj, K., Pattabiraman, K., and Mesbah, A. (2013). An empirical study of client-side javascript bugs. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 55–64. IEEE.

Offutt, A. J. and Untch, R. H. (2001). Mutation 2000: Uniting the orthogonal. In Wong, W. E., editor, *Mutation Testing for the New Century*, pages 34–44. Springer US, Boston, MA.

Petrović, G. and Ivanković, M. (2018). State of mutation testing at google. In Paulisch, F. and Bosch, J., editors, *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, pages 163–171, New York, NY, USA. ACM.

Sánchez, A. B., Delgado-Pérez, P., Medina-Bulo, I., and Segura, S. (2022). Mutation testing in the wild: findings from github. *Empirical Software Engineering*, 27(6).

vom Brocke, J., Simons, A., Niehaves, B., Niehaves, B., Reimer, K., Plattfaut, R., and Cleven, A. (2009). Reconstructing the giant: On the importance of rigour in documenting the literature search process. In Newell, S., Whitley, E., Pouloudi, N., Wareham, J., and Mathiassen, L., editors, *Information systems in a globalising world*, volume 161, Verona.