Do Null-Type Mutation Operators
Help Prevent Null-Type Faults?

Ali Parsai[1][0000−0001−8525−8198] and Serge Demeyer[2][0000−0002−4463−2945]

[1] University of Antwerp
`ali.parsai@uantwerpen.be`
[2] University of Antwerp and Flanders Make
`serge.demeyer@uantwerpen.be`

Abstract. The null-type is a major source of faults in Java programs, and its overuse has a severe impact on software maintenance. Unfortunately traditional mutation testing operators do not cover null-type faults by default, hence cannot be used as a preventive measure. We address this problem by designing four new mutation operators which model null-type faults explicitly. We show how these mutation operators are capable of revealing the missing tests, and we demonstrate that these mutation operators are useful in practice. For the latter, we analyze the test suites of 15 open-source projects to describe the trade-offs related to the adoption of these operators to strengthen the test suite.

Keywords: Software Maintenance, Software Testing, Mutation Testing, Null-Type, Test Quality

1   Introduction

The null-type is a special type in Java that has no name, cannot be casted, and practically equates to a literal that can be of any reference type [11]. The null-type is commonly misused, and frequently reported and discussed as an issue by developers [24]. The null-type is the source of the majority of faults in Java programs [25], and its overuse has a severe impact on software maintenance [15]. On the one hand, this scenario should push developers to build test suites capable of identifying null-type faults. On the other hand, developers without specific test requirements may struggle to identify all code elements or properties that the test must satisfy. To address this problem, we propose mutation testing as a way for improving the test suite to handle potential null-type faults.

Mutation testing is a technique to measure the quality of a test suite by assessing its fault detection capabilities [5]. Mutation testing is a two-step process. First, a small syntactic change is introduced in the production code. This change is obtained by applying a "mutation operator", and the resulting changed code is called a "mutant". Then, the test suite is executed for that mutant; if any of the tests fail, the mutant is "killed", otherwise, the mutant has "survived". Herein lies the aspect of mutation testing that we want to exploit: the identification of survived mutants that need to be killed.

Mutation operators are modeled after the common developer mistakes [14]. Over the years, multiple sets of mutation operators have been created to fit in different domains. By far the most commonly used mutation operators are the ones introduced in Mothra by Offutt et al. [21]. They use 10 programs written in Fortran to demonstrate that their reduced-set mutation operators is enough to produce a mutation-adequate test suite that can kill almost all of the mutants generated by the mutation operators of the complete-set. Later on, several attempts have been made to extend Offutt's mutation operators, for instance, to cope with the specificities of object-oriented programming [19]. Yet, none of the proposed mutation operators explicitly model null-type faults. As a result, mature general-purpose mutation testing tools currently used in literature, such as PITest [4] and Javalanche [35], do not cope explicitly with this type of faults by default. Therefore, the created mutants *risk* not being adequate to derive test requirements that handle null-type faults. Whether this risk is concrete or not depends on the ability of the available mutation operators to account for these faults. Yet, no study has explored this aspect.

This paper investigates the usefulness of mutation operators able to model null-type faults in order to strengthen the test suite against these faults. For this reason, we introduce four new mutation operators related to null-type faults. These mutation operators are modeled to cover the typical null-type faults introduced by developers [24]. We incorporate these mutation operators in LittleDarwin, an extensible open-source tool for mutation testing [32], creating a new version called LittleDarwin-Null. We organize our research in two steps: we show that (i) the current general-purpose mutation testing tools do not account for null-type faults by default, and modeling operators for null-type faults can drive the improvement of the test suite in practice, and (ii) the test suites of real open-source projects cannot properly catch null-type faults. The paper is driven by the following research questions:

- **RQ1:** *Are traditional mutation operators enough to prevent null-type faults?*
- **RQ2:** *To what extent is the addition of null-type mutation operators useful in practice?*

The rest of the paper is organized as follows: In Section 2, background information and related work is provided. In Section 3, the details of the experiment are discussed. In Section 4, the results are analyzed. In Section 5, we discuss the threats that affect the results. Finally, we present the conclusion in Section 6.

2   Background and Related Work

Mutation testing is the process of injecting faults into a software system and then verifying whether the test suite indeed fails, and thus detects the injected fault. First, a faulty version of the software is created by introducing faults into the system *(Mutation)*. This is done by applying a transformation *(Mutation Operator)* on a certain part of the code. After generating the faulty version of the software *(Mutant)*, it is passed onto the test suite. If a test fails, the mutant

is marked as killed *(Killed Mutant)*. If all tests pass, the mutant is marked as survived *(Survived Mutant)*.

Mutation Operators. A mutation operator is a transformation which introduces a single syntactic change into its input. The first set of mutation operators were reported in King et al. [16]. These mutation operators work on essential syntactic entities of programming languages such as arithmetic, logical, and relational operators. For object-oriented languages, new mutation operators were proposed [19]. The mature mutation testing tools of today still mostly use the *traditional* (i.e. method-level) mutation operators [27].

Equivalent Mutants. An *equivalent mutant* is a mutant that does not change the semantics of the program, i.e. its output is the same as the original program for any possible input. Therefore, no test case can differentiate between an equivalent mutant and the original program. The detection of equivalent mutants is undecidable due to the halting problem [22].

Mutation Coverage. Mutation testing allows software engineers to monitor the fault detection capability of a test suite by means of mutation coverage [13]. A test suite is said to achieve *full mutation test adequacy* whenever it can kill all the non-equivalent mutants, thus reaching a mutation coverage of 100%. Such test suite is called a *mutation-adequate test suite*.

Mutant Subsumption. Mutant subsumption is defined as the relationship between two mutants `A` and `B` in which `A` subsumes `B` if and only if the set of inputs that kill `A` is guaranteed to kill `B` [18]. The subsumption relationship for faults has been defined by Kuhn in 1999 [17]. Later on, Ammann et al. tackled the theoretical side of mutant subsumption [2] where they define *dynamic* mutant subsumption as follows: Mutant `A` dynamically subsumes Mutant `B` if and only if (i) `A` is killed, and (ii) every test that kills `A` also kills `B`. The main purpose behind the use of mutant subsumption is to detect redundant mutants. These mutants create multiple threats to the validity of mutation analysis [26]. This is done by determining the dynamic subsumption relationship among a set of mutants, and keep only those that are not subsumed by any other mutant.

Mutation Testing Tools. In this study, we use three different mutation testing tools: Javalanche, PITest, and LittleDarwin. Javalanche is a mutation testing framework for Java programs that attempts to be efficient, and not produce equivalent mutants [35]. It uses byte code manipulation in order to speed up the process of mutation testing. Javalanche has been used in numerous studies in the past (e.g. [10, 9]). PITest is a state-of-the-art mutation testing system for Java, designed to be fast and scalable [4]. PITest is the de facto standard for mutation testing within Java, and it is used as a baseline in mutation testing research (e.g. [12, 34]). LittleDarwin is a mutation testing tool designed to work out of the box with complicated industrial build systems. For this, it has a loose coupling with the test infrastructure, instead relying on the build system to run the test suite. LittleDarwin has been used in several studies, and is capable of performing mutation testing on complicated software systems [33, 30, 31]. For more information about LittleDarwin please refer to Parsai et al. [32]. We implemented the new null-type mutation operators in a special version of

LittleDarwin called LittleDarwin-Null. LittleDarwin and LittleDarwin-Null only differ in the set of mutation operators used, and are identical otherwise.

Related Work. Creating new mutation operators to deal with the evolution of software languages is a trend in mutation testing research. For example, mutation operators have been designed to account for concurrent code [3], aspect-oriented programming [7], graphical user interfaces [23], modern C++ constructs [29], and Android applications [6]. Nanavati et al. have previously studied mutation operators targeting memory-related faults [20]. However, the difference in the semantics of null object of Java and NULL macro of C is sufficient to grant the need for a separate investigation.

3   Experimental Setup

In this section, we first introduce our proposed mutation operators, and then we discuss the experimental setup we used to address our research questions.

3.1   Null-Type Mutation Operators

We derived four null-type mutation operators to model the typical null-type faults often encountered by developers [25]. These mutation operators are presented in Table 1.

Table 1. Null-Type Faults and Their Corresponding Mutation Operators

Mutation Operator	Description
NullifyReturnValue	If a method returns an object, it is replaced by `null`
NullifyInputVariable	If a method receives an object reference, it is replaced by `null`
NullifyObjectInitialization	Wherever there is a `new` statement, it is replaced with `null`
NegateNullCheck	Any binary relational statement containing `null` at one side is negated

3.2   Case Study

For RQ1, we use a didactic project. For RQ2, we use 15 open-source projects.

RQ1. In order to address RQ1, we chose a modified version of VideoStore as a small experimental project [8]. Choosing a small project allows us to (i) create a mutation-adequate test suite ourselves, (ii) find out which mutants are equivalent, and (iii) avoid complexities when using multiple mutation testing tools. The source code for VideoStore is available in the replication package.

RQ2. We selected 15 open-source projects for our empirical study (Table 2). The selected projects differ in size of their production code and test code, number of commits, and team size to provide a wide range of possible scenarios. Moreover, they also differ in the adequacy of their test suite based on statement, branch, and mutation coverage (Table 2). We used JaCoCo and Clover for statement and branch coverage, and LittleDarwin for mutation coverage.

4   Results and Discussion

RQ1: Are traditional mutation operators enough to prevent null-type faults?

We are interested to compute the number of killed, survived and equivalent mutants along with three versions of VideoStore. The first version we analyze

Table 2. Projects Sorted by Mutation Coverage

Project	Ver.	Size (LoC) Prod.	Size (LoC) Test	#C	TS	SC	BC	MC
Apache Commons CLI	1.3.1	2,665	3,768	816	15	96%	93%	94%
JSQLParser	0.9.4	7,342	5,909	576	19	81%	73%	94%
jOpt Simple	4.8	1,982	6,084	297	14	99%	97%	92%
Apache Commons Lang	3.4	24,289	41,758	4,398	30	94%	90%	91%
Joda Time	2.8.1	28,479	54,645	1,909	42	90%	81%	82%
Apache Commons Codec	1.10	6,485	10,782	1,461	10	96%	92%	82%
Apache Commons Collections	4.1	27,914	32,932	2,882	26	85%	78%	81%
VRaptor	3.5.5	14,111	15,496	3,417	65	87%	81%	81%
HTTP Request	6.0	1,391	2,721	446	15	94%	75%	78%
Apache Commons FileUpload	1.3.1	2,408	1,892	846	19	76%	74%	77%
jsoup	1.8.3	10,295	4,538	888	43	82%	72%	76%
JGraphT	0.9.1	13,822	8,180	1,150	31	79%	73%	69%
PITest	1.1.7	17,244	19,005	1,044	19	79%	73%	63%
JFreeChart	1.0.17	95,354	41,238	3,394	4	53%	45%	35%
PMD	r7706	70,767	43,449	7,706	20	62%	54%	34%

Acronyms: Version (Ver.), Line of code (LoC), Production code (Prod.), Number of commits (#C),
Team size (TS), Statement coverage (SC), Branch coverage (BC), Mutation coverage (MC)

is the original one (VideoStore Orig). This version has only 4 tests. Then, we create a mutation-adequate test suite that kills all mutants generated by the general-purpose tools (Javalanche, PITest, and LittleDarwin). In this version (VideoStore TAdq) we added 15 tests. Finally, we create a mutation-adequate test suite that kills all mutants, included the ones generate by LittleDarwin-Null. In this version (VideoStore NAdq) we added 3 more tests.

Table 3. Mutation testing results for VideoStore

Program	LittleDarwin			PITest			Javalanche			LittleDarwin-Null		
	K	S	E	K	S	E	K	S	E	K	S	E
VideoStore Orig	24	18	2	25	43	5	87	69	11	11	14	1
VideoStore TAdq	42	0	2	68	0	5	202	0	11	22	3	1
VideoStore NAdq	42	0	2	68	0	5	202	0	11	25	0	1

K: Killed, S: Survived, E: Equivalent

Table 3 shows the number of remaining mutants after each phase of test development: VideoStore Orig, VideoStore TAdq, and VideoStore NAdq. The discrepancy in total number of generated mutants for the three versions of the program in case of Javalanche is due to its particular optimizations. In Video-Store Orig, there are several survived mutants according to all the tools. This is because the test suite accompanying the VideoStore program was not adequate.

In VideoStore TAdq, we create a mutation-adequate version of the test suite with respect to the results of PITest, Javalanche, and LittleDarwin. In the process of creating this test suite, we noticed that all of these tools produce equivalent mutants. Two of such mutants are shown in Figure 2. Mutant A is equivalent because the method `super.determineAmount` always returns 0, so it does not matter whether it is added to or subtracted from `thisAmount`. Mutant B is also equivalent, because if

Fig. 1. The Surviving Non-Equivalent Null-Type Mutants

`daysRented` is 2, the value added to `thisAmount` is 0. We analyzed VideoStore TAdq with LittleDarwin-Null in order to find out whether the mutation-adequate test suite according to three general-purpose tools is able to kill all the null-type mutants. By analyzing the 26 generated mutants, we noticed that 22 mutants were killed and 4 survived. The manual review of these mutants show that one of them is an equivalent mutant.

Fig. 2. Two of the Equivalent Mutants Generated by Traditional Mutation Operators

Fig. 3. One of the Equivalent Mutants Generated by Null-Type Mutation Operators

Considering that 3 mutants generated by null-type mutation operators are not equivalent, and yet the mutation-adequate test suite we created according to the general-purpose tools cannot kill them, we conclude that **using traditional mutation operators to strengthen the test suite does not necessarily prevent null-type faults**.

The four mutants survived in VideoStore TAdq are all of type *NullifyObjectInitialization*. Figure 3 shows the equivalent null-type mutant. Here the default behavior of `Rental` object is to create a new `RegularMovie` object when it receives `null` as its input. So, replacing `new RegularMovie(null)` with null does not change the behavior of the program.

The three remaining surviving mutants are described in Figure 1. Here, mutants A and B replace the exception with `null`. Consequently, as opposed to the program throwing a detailed exception, the mutant always throws an empty `NullPointerException`. Such a mutant is desirable to kill, since the program would be able to throw an unexpected exception due to a fault that the test suite cannot recognize. In the case of Mutant C, it replaces the initialization of a `RegularMovie` object with `null`. This means that as opposed to the program that guarantees the private attribute `movie` is always instantiated, the same attribute contains a null literal in the mutant.

Fig. 4. The Tests Written to Kill the Surviving Null-Type Mutants

If not detected, a `NullPointerException` might be thrown when another object tries to access the `movie` attribute of this object.

We created three new tests to kill each of the survived mutants. These tests are shown in Figure 4. Here, `testMutantA` and `testMutantB` verify whether the unit under test throws the correct exception if called with an invalid input value. `testMutantC` verifies whether the unit under test is able to handle a null input correctly. These three tests are not "happy path tests", namely a well-defined test case using known input, which executes without exception and produces an expected output. Consequently, they might not be intuitive for a test developer to consider, even though they are known as good testing practice [1]. If not for the three survived null-type mutants, these tests would not have been written. This leads us to conclude that **traditional mutation operators are not enough to prevent null-type faults.**

RQ2: To what extent is the addition of null-type mutation operators useful in practice?

RQ1 shows for the VideoStore project that mutation testing tools need to introduce explicit mutation operators for modeling null-type faults. Yet, such a project is not representative of real projects. In this RQ, we want to verify to what extent null-type mutation operators are useful *in practice*. For this reason, we perform an experiment that involves real open-source projects. After introducing null-type mutants, two groups of mutants are affected: (i) survived mutants are the targets the developer needs during test development, (ii) killed mutants show the types of faults the test suite can already catch.

Considering this, we can justify the effort needed for extending mutation testing by incorporating null-type mutants only if: (i) the real test suites do not already kill most of the null-type mutants, (ii) the null-type mutants are not increasing redundancy by a large margin. Otherwise, the current mutation testing tools are already "good enough" for preventing null-type faults.

To verify to what extent the null-type mutants "do matter" when testing for null-type faults we analyze both killed and survived mutants:

In case of survived mutants, we analyze the number of survived mutants that each mutation operator generates for each project. We divide this analysis into two parts. First, we analyze survived mutants for null-type and traditional mutation operators. Second, we analyze each mutation operator individually to find out which one produces the most surviving mutants. This analysis shows whether the survived mutants produced by the null-type mutation operators are "enough" to drive the test development process.

In case of killed mutants, we take all projects as a whole, and we analyze whether the killed null-type mutants are redundant when used together with traditional mutation operators. We measure redundancy using dynamic mutant subsumption: we analyze the distributions of subsuming, killed, and all null-type mutants. This way we can tell whether or not the null-type mutation operators are producing "valuable" mutants to strengthen the test suite.

Survived mutants. Table 4 shows for each project the number of survived, killed, and total generated mutants for both groups of mutation operators. The first noticeable trend is a strong correlation ($R^2 = 0.81$) between survived to killed ratio (SKR) of the traditional mutants and SKR of the null-type mutants. One exception to this trend is JSQLParser, in which there are significantly more survived null-type mutants than survived traditional mutants. Investigating further, we find that this happens because 50 small classes lack statements that can be mutated by the traditional mutation operators. However, null-type mutation operators are able to generate mutants for these classes. This uncovers many of the weaknesses of the test suite. On the other side of the fence, there is PITest, in which a single class (`sun.pitest.CodeCoverageStore`) contains many arithmetic operations while poorly tested, so it produces 129 out of 398 survived traditional mutants. This shows that **the usefulness of the null-type mutation operators is program-dependent.**

Table 4. Mutants Generated by LittleDarwin and LittleDarwin-Null

Project	Traditional Mutation Operators			Null-Type Mutation Operators		
	Survived	Killed	Total	Survived	Killed	Total
Apache Commons CLI	24	318	342	71	415	486
JSQLParser	31	457	488	358	1,062	1,420
jOpt Simple	17	189	206	37	494	531
Apache Commons Lang	559	5,455	6,014	564	5,469	6,033
Joda Time	892	3,978	4,870	836	5,371	6,207
Apache Commons Codec	364	1,612	1,976	147	927	1,074
Apache Commons Collections	638	2,705	3,343	1,179	5,851	7,030
VRaptor	111	478	589	795	2,111	2,906
HTTP Request	49	178	227	69	383	452
Apache Commons FileUpload	81	273	354	137	211	348
jsoup	291	928	1,219	553	1,455	2,008
JGraphT	416	940	1,356	834	1,457	2,291
PITest	398	672	1,070	551	2,964	3,515
JFreeChart	10,558	5,603	16,161	8,563	6,248	14,811
PMD	5,205	2,734	7,939	5,099	4,613	9,712
Total	**19,634**	**26,520**	**46,154**	**19,793**	**39,031**	**58,824**

Figure 5 shows the number of killed and survived mutants for each mutation operator. We see that among the traditional mutation operators, *Arithmetic-OperatorReplacementBinary*, *LogicalOperatorReplacement*, and *ArithmeticOperatorReplacementUnary* have the highest ratio of survived to killed mutants. This means that these mutation operators are generating mutants that are harder to kill than the rest. The same can be observed among the null-type mutation operators, where *NullifyObjectInitialization* produces harder to kill mutants than the others. This is as we expected, since *NullifyInputVariable* applies a major change to the method (removal of an input), and *NegateNullCheck* negates a check that the developer deemed necessary. However, the unexpected part of the result is that so many of the mutants generated by *NullifyReturnValue* have survived. This means that lots of methods are not tested on their output correctly. This can be due to the fact that many of such methods are not tested directly, and when tested indirectly, their results only affect a small part of the program state of the method under test.

In general, the number of survived null-type mutants has a strong correlation with the number of survived traditional mutants for most projects. This implies that not all parts of the code are tested well. However, the exceptions to this rule are caused by classes that produce many more mutants of a particular type. Here, our results show that **the null-type mutation operators complement the traditional mutation operators and vice versa by each providing a large portion of survived mutants**.

Killed mutants. Considering all projects as a whole, the number of generated mutants is 104,978. Out of this total, the number of killed and subsuming mutants are 65,551 and 16,205 respectively. This means that at least 50,029 were subsumed, and thus redundant. To put null-type and traditional mutants in perspective, Figure 6 shows the percentages for all, killed, and subsuming mutants for both groups. Here, we notice that the percentage of the null-type mutants remains similar in these three categories. The null-type mutants have a higher
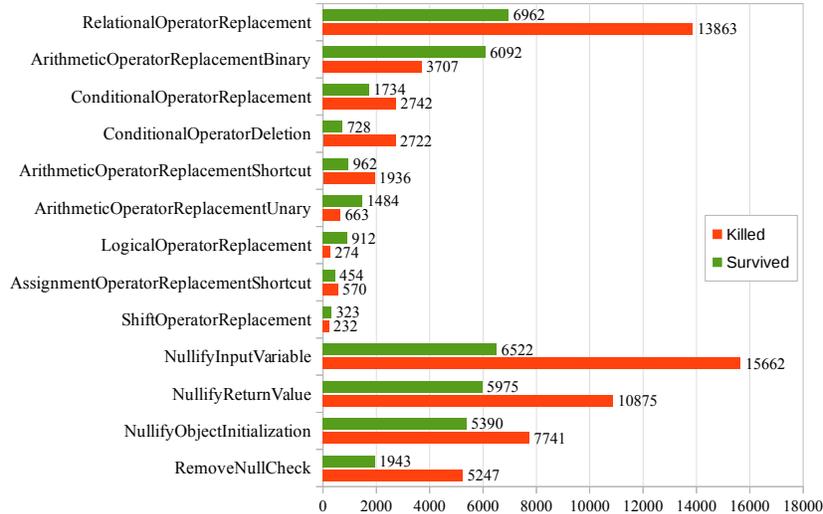
Fig. 5. Number of killed and survived mutants for each mutation operator

impact on the semantics of the program due to being applied at the entry and exit points of a method, the branching statements, and the declaration of an object. Therefore, the fact that they comprise a higher percentage of the killed mutants is not surprising. However, it is important to note that the distribution of null-type mutants differs only 4% in all and killed mutants. While 60% of the killed mutants are null-type, they still account for almost 55% of subsuming mutants. This indicates that **the inclusion of the null-type mutants increases the mutant redundancy only marginally**.

To delve deeper, Figure 7 shows for each mutation operator the percentage of killed and subsuming mutants. Among the traditional mutation operators, *RelationalOperatorReplacement* and *ConditionalOperatorReplacement* produce the most subsuming mutants. The rest of the mutation operators create mutants that have the same distribution among subsuming and killed mutants. As this figure shows, the marginal increase in

Fig. 6. Ratio of Null-Type and Traditional Mutants in All, Killed, and Subsuming

redundancy by the null-type mutation operators can be blamed on *NullifyInputVariable* mutation operator. This mutation operator produces mutants that are easier to kill compared to other mutation operators (21% of all, 24% of the killed), and more of these mutants are redundant compared to others (24% of killed, only 15% of subsuming). On the contrary, *NullifyReturnValue* is producing fewer redundant mutants, which confirms our previous observation.
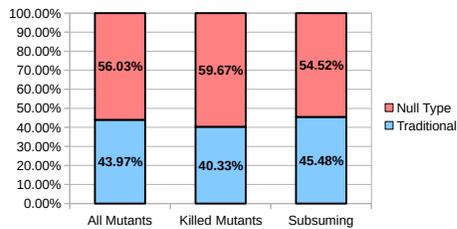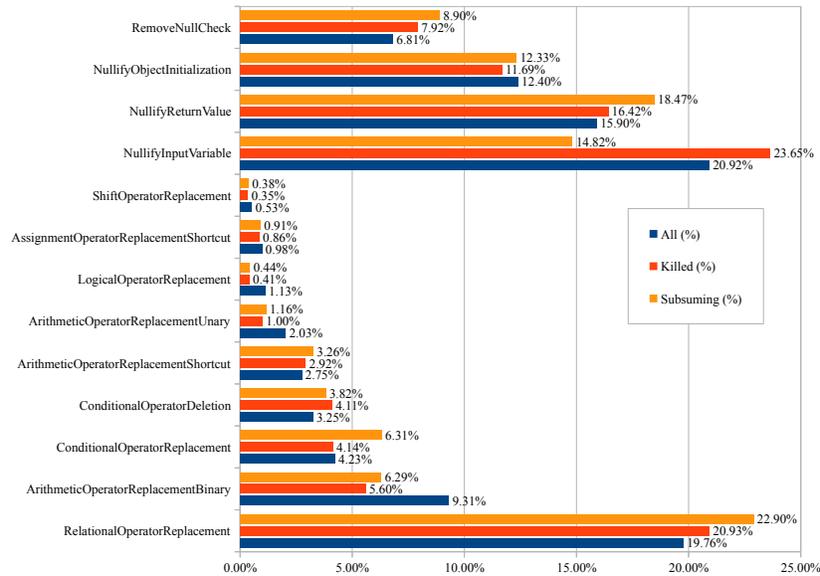
Fig. 7. Ratio of Mutants by Each Mutation Operator in All, Killed, and Subsuming

Given the results of RQ2, we can conclude that **while the inclusion of the null-type mutation operators increases the redundancy marginally, they complement the traditional mutation operators in their role of strengthening the test suite against null-type faults.**

5 Threats to Validity

To describe the threats to validity we refer to the guidelines reported by Yin [36]. Threats to **internal validity** focus on confounding factors that can influence the obtained results. These threats stem from potential faults hidden inside our analysis tools. While theoretically possible, we consider this chance limited. The tools used in this experiment have been used previously in several other studies, and their results went through many iterations of manual validation. In addition, the code of LittleDarwin and LittleDarwin-Null along with all the raw data of the study is publicly available for download in the replication package [28].

Threats to **external validity** refer to the generalizability of the results. In RQ1 we advocate for the adoption of null-type mutation operators by using a didactic project. We alleviate the non-representativeness of this project, by analyzing 15 real open-source projects in RQ2. Although our results are based on projects with various levels of test adequacy in terms of traditional and null-type mutation coverage, we cannot assume that this sample is representative of all Java projects. We use PITest, LittleDarwin, and Javalanche as mutation testing tools. We cannot assume that these tools are representative of all mutation tools available in literature. For this reason, we refer to these tools as general-purpose since they can work with little effort on many open-source projects. We modeled

null-types mutation operators upon the typical null-type faults described by Osman et al. [24]. However, there may be other types of null-type faults that we did not consider. Even if this was the case, our results should still hold since we already demonstrate with four mutation operators that they are in need of explicit modeling.

Threats to **construct validity** are concerned with how accurately the observations describe the phenomena of interest. The problem of equivalent mutants affects the analysis of surviving mutants on the test suites of the 15 open-source projects. Due to the large number of created mutants, it is impractical to filter equivalent mutants in the final results. Still, we believe this threat is minimal, because we analyze two different aspects of mutation testing, which lead to converging results. The total number of generated mutants can be different based on the set of mutation operators that are used in each tool. However, this difference has been taken into account when discussing the results of the experiments. To measure redundancy among the mutants, we use dynamic subsumption relationship. However, the accuracy of the dynamic subsumption relationship depends on the test suite itself. This is a compromise, as the only way to increase the accuracy is to have several tests that kill each mutant, which is not practical.

6   Conclusion

Developers are prone to introduce null-type faults in Java programs. Yet, there is no specific approach devoted to helping developers strengthen the test suite against these faults. On the one hand, mutation testing provides a systematic method to create tests able to prevent common faults. On the other hand, the general-purpose mutation testing tools available today do not model null-type faults explicitly by default.

In this paper, we advocate for the introduction of null-type mutation operators for preventing null-type faults. As a first step, we show that traditional mutation operators are not enough to cope with null-type faults as they cannot lead to the creation of a mutation-adequate test suite that can kill all of them. Then we demonstrate, by means of code examples, how the null-type mutants can drive the extension of the test suite. Finally, we highlight that null-type mutation operators are helpful in practice by showing on 15 open-source projects that real test suites are inadequate in detecting null-type faults. In this context, we explore the trade-offs of having null-type mutants. On the downside, we show that the inclusion of null-type mutants increases the mutant redundancy. Yet, this increment is only marginal. On the upside, we show that null-type mutants complement traditional mutants in two ways. First, they provide a large number of survived mutants to the developer to strengthen the test suite. Second, they comprise a large part of subsuming mutants.

As a consequence, developers can increase their confidence in the test suite regarding to the null-type faults by (i) prioritizing the classes that have a large difference in traditional and null-type mutation coverage, (ii) creating tests to kill the survived null-type mutants in these classes, and (iii) repeating the process until all classes have similar levels of traditional and null-type mutation coverage.

References

1. Alexander, I.: Misuse cases: use cases with hostile intent. IEEE Software **20**(1), 58–66 (jan 2003). https://doi.org/10.1109/ms.2003.1159030
2. Ammann, P., Delamaro, M.E., Offutt, J.: Establishing theoretical minimal sets of mutants. In: 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation. pp. 21–30. IEEE (mar 2014). https://doi.org/10.1109/icst.2014.13
3. Bradbury, J.S., Cordy, J.R., Dingel, J.: Mutation operators for concurrent java (J2SE 5.0). In: Second Workshop on Mutation Analysis (Mutation 2006 - ISSRE Workshops 2006). pp. 11—-. MUTATION '06, Sch. of Comput., Queen"s Univ., Kingston, ON, IEEE, Washington, DC, USA (nov 2006). https://doi.org/10.1109/mutation.2006.10
4. Coles, H., Laurent, T., Henard, C., Papadakis, M., Ventresque, A.: PIT: a practical mutation testing tool for java (demo). In: Proceedings of the 25th International Symposium on Software Testing and Analysis - ISSTA 2016. pp. 449–452. ISSTA 2016, ACM Press, New York, NY, USA (2016). https://doi.org/10.1145/2931037.2948707
5. DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Hints on test data selection: Help for the practicing programmer. Computer **11**(4), 34–41 (apr 1978). https://doi.org/10.1109/c-m.1978.218136
6. Deng, L., Offutt, J., Ammann, P., Mirzaei, N.: Mutation operators for testing android apps. Information and Software Technology **81**, 154–168 (jan 2017). https://doi.org/10.1016/j.infsof.2016.04.012
7. Ferrari, F.C., Maldonado, J., Rashid, A.: Mutation testing for aspect-oriented programs. In: 2008 International Conference on Software Testing, Verification, and Validation. pp. 52–61. ICST '08, Dept. of Comput. Syst., Sao Paulo Univ., Sao Carlos, IEEE, Washington, DC, USA (apr 2008). https://doi.org/10.1109/icst.2008.37
8. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Boston, MA, USA (1999)
9. Fraser, G., Zeller, A.: Mutation-driven generation of unit tests and oracles. IEEE Transactions on Software Engineering **38**(2), 278–292 (mar 2012). https://doi.org/10.1109/tse.2011.93
10. Gligoric, M., Groce, A., Zhang, C., Sharma, R., Alipour, M.A., Marinov, D.: Comparing non-adequate test suites using coverage criteria. In: Proceedings of the 2013 International Symposium on Software Testing and Analysis - ISSTA 2013. pp. 302–313. ISSTA 2013, ACM Press, New York, NY, USA (2013). https://doi.org/10.1145/2483760.2483769
11. Gosling, J., Joy, B., Steele, G., Bracha, G., Buckley, A.: The Java Language Specification (Java SE 8 edition). Oracle, java se 8 edn. (2014)
12. Inozemtseva, L., Holmes, R.: Coverage is not strongly correlated with test suite effectiveness. In: Proceedings of the 36th International Conference on Software Engineering - ICSE 2014. pp. 435–445. ICSE 2014, ACM Press, New York, NY, USA (2014). https://doi.org/10.1145/2568225.2568271
13. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Transactions on Software Engineering **37**(5), 649–678 (sep 2011). https://doi.org/10.1109/tse.2010.62
14. Just, R., Jalali, D., Inozemtseva, L., Ernst, M.D., Holmes, R., Fraser, G.: Are mutants a valid substitute for real faults in software testing? In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software

Engineering - FSE 2014. pp. 654–665. FSE 2014, ACM Press, New York, NY, USA (2014). https://doi.org/10.1145/2635868.2635929

15. Kimura, S., Hotta, K., Higo, Y., Igaki, H., Kusumoto, S.: Does return null matter? In: 2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE). pp. 244–253. IEEE (feb 2014). https://doi.org/10.1109/csmr-wcre.2014.6747176

16. King, K.N., Offutt, A.J.: A fortran language system for mutation-based software testing. Software: Practice and Experience **21**(7), 685–718 (jul 1991). https://doi.org/10.1002/spe.4380210704

17. Kuhn, D.R.: Fault classes and error detection capability of specification-based testing. ACM Transactions on Software Engineering and Methodology **8**(4), 411–424 (oct 1999). https://doi.org/10.1145/322993.322996

18. Kurtz, B., Ammann, P., Offutt, J.: Static analysis of mutant subsumption. In: 2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW). pp. 1–10. IEEE (apr 2015). https://doi.org/10.1109/icstw.2015.7107454

19. Ma, Y.S., Kwon, Y.R., Offutt, J.: Inter-class mutation operators for java. In: 13th International Symposium on Software Reliability Engineering, 2002. Proceedings. pp. 352–363. IEEE Comput. Soc (2002). https://doi.org/10.1109/issre.2002.1173287

20. Nanavati, J., Wu, F., Harman, M., Jia, Y., Krinke, J.: Mutation testing of memory-related operators. In: 2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW). pp. 1–10. IEEE (apr 2015). https://doi.org/10.1109/icstw.2015.7107449

21. Offutt, A.J., Lee, A., Rothermel, G., Untch, R.H., Zapf, C.: An experimental determination of sufficient mutant operators. ACM Transactions on Software Engineering and Methodology **5**(2), 99–118 (apr 1996). https://doi.org/10.1145/227607.227610

22. Offutt, A.J., Pan, J.: Automatically detecting equivalent mutants and infeasible paths. Software Testing, Verification and Reliability **7**(3), 165–192 (sep 1997). https://doi.org/10.1002/(sici)1099-1689(199709)7:3¡165::aid-stvr143¿3.0.co;2-u

23. Oliveira, R.A., Alegroth, E., Gao, Z., Memon, A.: Definition and evaluation of mutation operators for GUI-level mutation analysis. In: 2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW). pp. 1–10. IEEE (apr 2015). https://doi.org/10.1109/icstw.2015.7107457

24. Osman, H., Leuenberger, M., Lungu, M., Nierstrasz, O.: Tracking null checks in open-source java systems. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER). vol. 1, pp. 304–313. IEEE (mar 2016). https://doi.org/10.1109/saner.2016.57

25. Osman, H., Lungu, M., Nierstrasz, O.: Mining frequent bug-fix code changes. In: 2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE). pp. 343–347. IEEE (feb 2014). https://doi.org/10.1109/csmr-wcre.2014.6747191

26. Papadakis, M., Henard, C., Harman, M., Jia, Y., Traon, Y.L.: Threats to the validity of mutation-based test assessment. In: Proceedings of the 25th International Symposium on Software Testing and Analysis - ISSTA 2016. pp. 354–365. ISSTA 2016, ACM Press, New York, NY, USA (2016). https://doi.org/10.1145/2931037.2931040

27. Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Traon, Y.L., Harman, M.: Mutation testing advances: An analysis and survey. Advances in Computers (2018). https://doi.org/10.1016/bs.adcom.2018.03.015

28. Parsai, A.: Replication package, http://parsai.net/files/research/SofSemReplicationPackage.7z
29. Parsai, A., Demeyer, S., Busser, S.D.: C++11/14 mutation operators based on common fault patterns. In: Medina-Bulo, I., Merayo, M.G., Hierons, R. (eds.) Testing Software and Systems, pp. 102–118. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-99927-2_9
30. Parsai, A., Murgia, A., Demeyer, S.: Evaluating random mutant selection at class-level in projects with non-adequate test suites. In: Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering - EASE 2016. pp. 11:1–11:10. EASE '16, ACM Press, New York, NY, USA (2016). https://doi.org/10.1145/2915970.2915992
31. Parsai, A., Murgia, A., Demeyer, S.: A model to estimate first-order mutation coverage from higher-order mutation coverage. In: 2016 IEEE International Conference on Software Quality, Reliability and Security (QRS). pp. 365–373. IEEE (aug 2016). https://doi.org/10.1109/qrs.2016.48
32. Parsai, A., Murgia, A., Demeyer, S.: LittleDarwin: A feature-rich and extensible mutation testing framework for large and complex java systems. In: Fundamentals of Software Engineering: 7th International Conference, FSEN 2017, Tehran, Iran, April 26–28, 2017, Revised Selected Papers, pp. 148–163. Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-68972-2_10
33. Parsai, A., Murgia, A., Soetens, Q.D., Demeyer, S.: Mutation testing as a safety net for test code refactoring. In: Scientific Workshop Proceedings of the XP2015 on - XP 2015 workshops. pp. 8:1–8:7. XP '15 workshops, ACM Press, New York, NY, USA (2015). https://doi.org/10.1145/2764979.2764987
34. Parsai, A., Soetens, Q.D., Murgia, A., Demeyer, S.: Considering polymorphism in change-based test suite reduction. In: Dingsøyr, T., Moe, N.B., Tonelli, R., Counsell, S., Gencel, C., Petersen, K. (eds.) Lecture Notes in Business Information Processing, pp. 166–181. Springer International Publishing, Cham (2014). https://doi.org/10.1007/978-3-319-14358-3_14
35. Schuler, D., Zeller, A.: Javalanche: efficient mutation testing for java. In: Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering on European software engineering conference and foundations of software engineering symposium - ESEC/FSE 2009. pp. 297–298. ESEC/FSE '09, ACM Press, New York, NY, USA (2009). https://doi.org/10.1145/1595696.1595750
36. Yin, R.K.: Case Study Research: Design and Methods. Applied Social Research Methods, SAGE Publications (2003)