SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 0000; **00**:1–28
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr

Performance Mutation Testing

Pedro Delgado-Pérez[1]*, Ana B. Sánchez[2], Sergio Segura[2], Inmaculada Medina-Bulo[1]

[1] *Universidad de Cádiz, Escuela Superior de Ingeniería, Spain.*
[2] *Universidad de Sevilla, ETS Ingeniería Informática, Spain.*

SUMMARY

Performance bugs are known to be a major threat to the success of software products. Performance tests aim to detect performance bugs by executing the program through test cases and checking whether it exhibits a noticeable performance degradation. The principles of mutation testing, a well-established testing technique for the assessment of test suites through the injection of artificial faults, could be exploited to evaluate and improve the detection power of *performance tests*. However, the application of mutation testing to assess performance tests, henceforth called *performance mutation testing* (PMT), is a novel research topic with numerous open challenges. In previous papers, we identified some key challenges related to PMT. In this work, we go a step further and explore the feasibility of applying PMT at the source-code level in general-purpose languages. To do so, we revisit concepts associated with classical mutation testing, and design seven novel mutation operators to model known bug-inducing patterns. As a proof of concept, we applied traditional mutation operators as well as performance mutation operators to open-source C++ programs. The results reveal the potential of the new performance-mutants to help assess and enhance performance tests when compared to traditional mutants. A review of live mutants in these programs suggests that they can induce the design of special test inputs. In addition to these promising results, our work brings a whole new set of challenges related to PMT, which will hopefully serve as a starting point for new contributions in the area. Copyright © 0000 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: Software testing, mutation testing, performance testing, performance bugs.

1. INTRODUCTION

Performance bugs are programming errors that can cause a significant performance degradation like excessive memory consumption [1] or energy leaks [2, 3]. Performance bugs affect to key non-functional properties of programs such as execution time or memory consumption. These types of bugs are very harmful and common in released software programs [4], causing poor usability and waste of resources. This might lead to a loss of users, loss of money or, in the worst case, loss of human lives, e.g., a self-driving car not braking in time. For example, recent studies [5, 6] show that

*Correspondence to: Escuela Superior de Ingeniería, Avda. de la Universidad de Cádiz 10, 11519 Campus Universitario de Puerto Real, Spain. Email: pedro.delgado@uca.es

40% of the users will leave a website that takes more than 3 seconds to load, 8% of them will stop a purchase in a poor performing site, or 79% will be less likely to purchase again on the same site if they are dissatisfied with their visit in terms of performance. Therefore, performance bugs require full attention to increase the loyalty of users and the safety of systems.

Compared to functional faults, performance bugs are significantly harder to detect and require more time and effort to be fixed [3]. This is partly due to the lack of *test oracles*, that is, mechanisms to decide whether the performance of the program with a given input is acceptable i.e., the *oracle problem* [7, 8]. For instance, Nistor et al. [3] analyzed 210 performance bugs from three mature open source projects and concluded that "better oracles are needed for discovering performance bugs". In contrast to functional bugs, performance bugs do not usually produce wrong results or crashes in the program under test and therefore they cannot be detected by simply inspecting the program output. Furthermore, performance indicators are by nature non-deterministic and can vary among executions due to numerous factors, such as the device hardware, configuration settings, or the current workload [9]. For example, suppose a mobile shopping app that consumes 30Mb of memory: Is this the expected performance? How much memory should it be considered as a performance bug? These types of questions are typically answered by using execution profilers [1, 10, 11] or performance tests [12, 13]. Performance bugs can also be detected by searching for bug-inducing patterns identified in previous versions of the target or related programs [7, 14, 15, 16]. Applying performance tests is crucial to gain confidence that the developed systems are free of performance faults, that is, deviations with respect to the specification in terms of performance. Note that this is different from program optimization [2, 17], whose goal is to improve the performance of a program to achieve a less consuming version. Despite the relevance of this issue, we have identified a lack of mechanisms to guide the assessment and improvement of performance tests.

Mutation testing is a fully-fledged testing technique for the assessment and improvement of test suites. In mutation testing, multiple faulty versions of the target program, so-called *mutants*, are generated by applying mutation operators. Each mutant is generated by applying a syntactic change to the original program. Those mutants are then executed against the test suite to evaluate its fault-revealing capability. Some mutants may not change the semantics of the program and are called semantically-equivalent mutants. Telling whether a mutant is equivalent is an undecidable problem, which currently represents one of the major obstacles for the wide-spread adoption of mutation testing [18]. Catalogs of mutation operators have been proposed in most popular programming languages. However, mutation testing has mostly been applied for functional testing, remaining its applicability to performance testing as a promising open research topic. Previous papers that relate mutation testing and performance have defined domain-specific mutation operators in Android [19] or model-based mutation operators [20, 21, 22]. There is also some incipient work in studying the applicability of traditional mutation operators to address non-functional properties [23]. We only found a paper that introduces the idea of using mutations to derive test suites for estimating the worst-case execution time [24]. To the best of our knowledge, the classical approach of mutation testing has not yet been applied with the intention of injecting mutations at the source-code level that affect the performance of the program.

In previous short papers [25, 26], we have explored the potential benefits and limitations of the application of mutation testing at the performance level, what we called *performance mutation testing* (PMT). Analogously to classic mutation testing, the goal of PMT is to generate variants of

DOI: 10.1002/stvr

the original program (performance-mutants), where each variant models a performance bug, e.g. one that causes a noticeable degradation of the execution time of the program. These artificial faulty programs could then be used to assess and improve the effectiveness of performance tests. However, our initial findings suggest that PMT presents unique problems that make it far more challenging than mutation testing at the functional level. For example, contrary to functional mutation testing, mutants in PMT should exhibit a certain performance degradation while preserving the semantics of the original program. Interestingly, this means that semantically-equivalent mutants are actually the target of PMT, rather than an obstacle, as in traditional mutation. This could lead to problems with compiling optimization techniques, for example, which could automatically undo performance mutations if they detect that the original program and the mutant are equivalent [27].

In this article, we give a step forward in studying the feasibility of PMT. In particular, we first propose seven performance mutation operators based on classical performance bug patterns identified in the literature [7, 1, 10, 15, 16, 17, 19, 28, 29]. Then, we evaluate their effectiveness in assessing the fault-detection capability of test suites in three real-world C++ programs. This required us to revisit the concept of valid, killed, alive and equivalent mutants. Among other results, we found that the new operators present greater potential to model useful performance faults than traditional operators. In particular, over 80% of performance-mutants generated in the programs under test (194 out of 241) were not detected by the functional test suite accompanying these applications. This is a positive aspect for performance-mutants because, by definition, mutants used in PMT are expected to preserve the internal state of the program. In contrast, the percentage of traditional mutants that are not detected by the functional test suite is much lower (42.7%), which means that more than half of traditional mutants are directly discarded when applying PMT. We illustrate the usefulness of performance-mutants with specific examples extracted from the code of the programs under test. Finally, we give evidence that most of the proposed mutations are not invalidated by the optimizations implemented in compilers, a necessary condition for the feasibility of this approach.

The structure of the paper is as follows. Section 2 reviews the state of the art with regard to performance bugs and mutation testing. Section 3 motivates our work describing the challenges in PMT and Section 4 presents the research questions. The definition of performance mutation operators is addressed in Section 5. Section 6 describes the experimental evaluation and Section 7 shows and discusses respectively the results of the experiments to answer the research questions. Section 8 and 9 respectively present threats to validity and surveys related work focused on mutation testing and non-functional properties. Finally, Section 10 presents the conclusions and pending research questions to tackle in the future.

2. BACKGROUND

In the following subsections, we briefly describe the main terms used throughout the article.

2.1. Performance bugs

A performance bug arises when a program produces the correct result but its behavior suffers a noticeable degradation in terms of performance [7, 16]. Therefore, a performance bug can be defined as a programming error that causes a significant performance degradation mainly due to the overuse

DOI: 10.1002/stvr

of computational resources like execution time [30], memory [1] and energy [2, 3]. As an example, consider the real performance bug found in Groovy, illustrated in Figure 1. Lines 4-7 are only meaningful when the array $argTypes$ is null or empty. However, those lines are always executed, wasting computation resources. To fix this, an *if* condition was added in line 3 to avoid lines 4-7 being executed unnecessarily [29].

According to the work by Jin et al. [7], there is a variety of common root causes that frequently lead to performance bugs, which include the inefficient combination of function calls, the unnecessary use of functions in certain contexts or synchronization issues. Well-tested applications such as Microsoft SQLServer, Apache HTTPD, and Mozilla Firefox, among others, are affected by hundreds of performance bugs [4, 3]. Performance bugs can cause latency, a security breach in software systems and a disproportionate waste of computation resources. For these reasons, we have to pay special attention to detecting these kinds of issues and refactoring the code to achieve a version that meets the performance requirements of our system.

There exist widely-used techniques to detect and fix performance problems such as the supervised execution with profilers [1, 10, 11] or the use of bug detection strategies that exploit known root causes for the appearance of performance bugs [7, 14, 15, 16]. However, performance bugs have usually been overlooked in traditional testing because they are more difficult to expose and detect than functional bugs since the former ones do not usually cause fail-stop symptoms [7]. The first meaningful studies on performance testing date back to the 90s [12]. Even though the interest in this field is growing in the last years [13, 28, 31, 32, 33], there is a lack of mechanisms to assess and improve the effectiveness of performance tests and it is acknowledged that most sophisticated performance testing strategies are required [8]. Considering all the above performance aspects, the detection of performance bugs presents numerous challenges to be addressed.

```
1     Class[] argTypes = ...
2     for (Iterator i = methods.iterator(); i.hasNext();) {
3         + if (!(argTypes == null) && !(argTypes.length == 0)) break;// FIX
4         MethodNode mn = (MethodNode) i.next();
5         boolean isZeroArg = (argTypes == null || argTypes.length == 0);
6         boolean match =mn.getName().equals(methodName) && isZeroArg;
7         if (match) return true;
8     }
```

Figure 1. Real performance bug found in Groovy [29].

2.2. *Mutation testing*

Mutation testing is a structural testing method that aims to assess and improve the efficacy of test suites in discovering faults [18]. To do that, this technique injects faults (i.e., mutations) through a number of predefined rules (i.e., mutation operators) that help us to measure the adequacy of a test suite detecting them. Each of the new versions of the program, known as *mutants*, are then executed against the test suite to produce an output, in the same way as the original program. When a test case detects a mutant (i.e., the output of the mutant differs from the output of the original program), it is said that the mutant has been *killed*. Otherwise, the mutant remains *alive*. Ideally, a good mutation operator should produce mutants which are not easy to detect so that they can reveal weaknesses in a test suite. In contrast, it should minimize the generation of *equivalent* mutants (i.e., there is no

Prepared using stvrauth.cls
DOI: 10.1002/stvr

input that can reveal the mutation). The problem of equivalent mutant identification together with the great number of mutants that can be produced are two stumbling blocks to a wider application of mutation testing.

This testing technique has been used in the past for testing software at different levels. Mutation testing has been applied to many programming languages at unit testing level (e.g., Java [34], C++ [35], C# [36] or PHP [37], among many others), and has also been used at the integration level [38]. Additionally, we can find several works that propose the use of mutation testing at the specification level in applications such as web services [39], finite state machines [40] or timed automata [33, 41].

3. PROBLEM STATEMENT

The problem behind *performance mutation testing* (PMT) can be defined as finding a set of mutation operators that allow to assess and improve the fault-detection effectiveness of performance tests. Each mutation operator should perform a change in the source code of the program that causes a noticeable performance degradation while preserving the functionality of the original program. In this work, we transplant the application of traditional mutation testing to this new domain in order to assess the quality of performance tests: just like functional tests assess the compliance of the program with the functional requirements, performance tests indicate how close the program is to satisfy the non-functional specification. Since such specifications are not typically available, performance tests assess if the program under test deviates from the expected performance i.e., observing a significant performance degradation. Hence, the goal is to create faulty versions of the program which are close to the correct version of the program, in line with the two classic hypotheses behind mutation testing: the Competent Programmer Hypothesis and the Coupling Effect [42]. The former, also called Finite Neighborhood Hypothesis [43], assumes that programmers are competent, and therefore they tend to write programs close to the ideal version. In this new domain, it is important to clarify that this ideal version is not the optimal program in terms of performance (where program optimization techniques may be helpful) but one that does show the expected performance. The latter, i.e., the Coupling Effect, states that tests that are adequate to identify bugs introduced by simple changes should also be able to detect more complex faults. In this article, we hypothesize that these principles hold for PMT as they do for functional mutation testing on the basis of existing works in the literature describing performance bugs. For instance, Jin et al. [7] found that most of the studied performance bugs could be fixed through simple changes.

It could be argued that a simple way to introduce performance bugs is to insert harmless delays (e.g., *Thread.sleep(100)*) or to force unnecessary resource consumption (e.g., generation of unused objects) at convenient points in the program. However, although such a naive approach may be helpful in certain cases, it may not represent the types of faults introduced by programmers. Jin et al. [7] conducted an empirical study of 109 real-world performance bugs and concluded that almost 50% of performance bugs require both inputs with special characteristics and large-scale inputs to manifest. Similarly, Liu et al. [15] studied 70 real-world performance bugs in Android applications and found that bugs required a certain sequence of inputs to be perceived. Therefore, we assume

DOI: 10.1002/stvr

that the performance bugs introduced with mutation should ideally mimic typical performance bugs made by programmers rather than simply synthetically consuming resources along the program.

In what follows, we define the classical terms related to mutation testing used throughout the article from a performance testing perspective, namely:

1. **Performance-mutant**. A variant of the original program where a syntactic change is introduced in the code to degrade the performance of the program. Those performance-mutants preserving the semantics of the original program are referred to as *valid* performance-mutants, while those changing the functionality will be labeled as *invalid*. The execution of the functional test suite on a performance-mutant is not a sufficient condition to determine whether that mutant is valid because the used test suite might not be complete enough to reveal a change in the functionality. Therefore, a performance-mutant that is not detected by the functional test suite is referred throughout the article to as *potentially valid*. Only a manual review can determine this condition with certainty.

2. **Killed performance-mutant**. A performance-mutant is killed if the tests can detect a noticeable performance degradation with respect to the performance observed in the original program. In this article, we use different thresholds to decide when a mutant is killed (cf. Section 6.2).

3. **Alive performance-mutant**. A performance-mutant is alive if it is not detected by our performance tests, but it is possible to design a *feasible test input* that uncovers the performance bug. By feasible we mean that the test input used to reveal the mutation is realistic for the program under test in a production environment. For instance, it could be the case that a performance mutation increased the execution time each time a line was read from a file: the more lines, the higher the degradation. However, that file may have a fixed number of lines not large enough to reveal the performance degradation in practice, and that situation should be assessed by the tester.

4. **Equivalent performance-mutant**. A performance-mutant is equivalent if there is no test input that can detect a noticeable increment between the observed performance in the original program and the mutant. Again, this definition depends on two factors: the thresholds and the feasibility of test inputs.

5. **Performance mutation score**. This metric allows assessing the fault-revealing capability of performance tests, measured as the ratio of killed performance-mutants to non-equivalent performance-mutants. By increasing this score, we improve the quality of the test suite and gain confidence that our software is free of performance bugs.

4. RESEARCH QUESTIONS

In this paper, we seek to evaluate the viability to apply PMT by defining domain-independent mutation operators (therefore, applicable to most programming languages) to assess and improve non-functional test suites. Namely, we aim at answering the following research questions in relation to PMT:

Prepared using stvrauth.cls
DOI: 10.1002/stvr

> ***Research question 1:*** *How feasible is to define domain-independent performance mutation operators?*

In this study, we aim at identifying some useful bug patterns to lead to plausible performance bugs (i.e., patterns that can potentially degrade the performance of the program), and studying their applicability in real-world programs.

Some recent papers have suggested the possibility of using traditional mutation operators in relation to performance [23, 25, 44]. By introducing traditional operators in our study, we want to shed light on whether these standard operators can model performance bugs or there is an actual need to incorporate new operators. To study the feasibility of using traditional operators for PMT, we address the following research question:

> ***Research question 2:*** *What percentage of mutants generated by traditional operators can model performance bugs?*

Even if traditional operators produce effective mutants for our purpose, it is interesting to know whether the new mutation operators focused on performance are more effective than their traditional counterparts. Therefore, we aim to answer the following question:

> ***Research question 3:*** *How many performance-mutants do performance mutation operators generate and how often do these mutants change the functional behavior?*

Once those performance-mutants have been executed against the test suite, we want to know whether those valid performance-mutants are able to cause a perceptible performance degradation and the kind of test inputs required to detect them. This is the aim of the next question:

> ***Research question 4:*** *Are the defined performance operators able to model some noticeable performance degradation?*

Finally, in a previous work we noted the connection between PMT and the optimizations implemented in compilers. More specifically, we observed that both compiler optimizations and performance mutations seek to preserve the functionality of the program and, therefore, the transformations carried out by the former could potentially undo the latter. As a result, we complete the study with the following question:

> ***Research question 5:*** *Can optimizations performed by compilers revert the mutations injected with performance mutation operators?*

5. PERFORMANCE MUTATION OPERATORS

In order to address **RQ1**, we first looked for real performance bug patterns that would inspire us to define performance-related mutation operators. Thus, we describe a set of mutation operators that

DOI: 10.1002/stvr

exploit common causes of performance degradation observed in the literature. We also studied the conditions under which these operators can be applied in order to generate valid mutants.

To reduce the complexity of the problem, this feasibility study focuses on general-purpose language features and non-concurrent programs (mutations could lead to synchronization issues). In particular, we defined a total of seven performance mutation operators, all of them affecting basic programming features which are present in most general-purpose programming languages (e.g., loops, conditional statements, containers, objects, etc.). In general, it should be possible to adapt these mutation operators to the concrete syntax of each specific language. Indeed, some of these operators have been defined on the basis of domain-specific operators (e.g., Android applications).

In most cases, the application of traditional mutation testing is straightforward as we only have to circumvent the generation of invalid mutants (i.e., mutants that are not grammatically correct). This fact is different in the case of PMT since, as previously explained, performance mutants should degrade one or more non-functional properties of the program under test while preserving its functionality. In order to achieve both objectives at the same time, it is often necessary to *preprocess* the source code. This can help prevent trivial situations that are known to alter the program semantics. To illustrate this, consider the example in Figure 2 of a performance-mutant generated by the mutation operator *Swap of Operands in Condition* or *SOC* (later defined in Section 5.4). This operator swaps the order of the conditions in a conditional statement with the aim of forcing to always evaluate the most time-consuming condition.

```
Original code:                          Mutant code:

if(obj && obj->method()){               if(obj->method() && obj){
    ...                                     ...
}                                       }
```

Figure 2. Example of *SOC* mutant.

As shown, this mutation will cause a functional issue when *obj* is *null* since we cannot invoke a method on a null object. We tackle this issue by means of the definition of proper *preconditions* for our mutation operators. It is important to remark at this point that the preconditions seek to avoid the injection of performance mutations that modify the functionality of the program, but we cannot always guarantee that the semantics is preserved.

For each mutation operator, we next describe its motivation, definition, an example, and a set of preconditions for its application. Table I summarizes the main characteristics of the mutation operators that have been defined.

5.1. RCL: Removal of Stop Condition in Loop

Motivation: According to Jin et al. [7] a *Skippable function* bug pattern happens when a function is invoked but it is unnecessary given the calling context. Similarly, sometimes we could configure a loop to stop when a certain condition is met, thus saving some unnecessary iterations that do not affect the outcome of the program. With regard to this, Nistor et al. [29] identified a family of performance bugs associated with a loop and a condition that have *CondBreak fixes*, i.e., bugs that

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Table I. Summary of performance mutation operators: source, abbreviation, category or main language features targeted, main performance property affected and need for preconditions.

Source	Operator	Category	Performance issue	Preconditions
[7, 29]	**RCL**	Loop perturbation	Execution time	✓
[16]	**URV**	Method call	Execution time	✓
[1, 10]	**MSL**	Object generation; Conditional statement; Loop perturbation	Memory consumption Execution time	✓
[7]	**SOC**	Conditional statement	Execution time	✓
[19, 28]	**HWO**	Method call	Execution time	
[10, 15]	**CSO**	Object generation	Memory consumption	
[10, 17]	**MSR**	Collections	Memory consumption	

can be fixed by adding a condition inside the loop. Figure 1 shown in Section 2 depicts an example of this kind of performance bug.

Definition: Inspired by this bug pattern, the *RCL* operator removes a stop condition in a loop so that the loop keeps iterating until another condition is satisfied. This is achieved by different means, namely 1) removing a *break* statement, 2) delaying a *return* statement with true or false, 3) removing a condition that uses a *boolean* variable expressly defined to stop the execution of the loop when the stop condition is met.

Example: The following code snippet shows the third option to apply the *RCL* operator. The mutant removes the second condition (*!b*) from the *while* loop, making the program to iterate while the first condition evaluates to true ($i < n$). However, the loop could skip some iterations in the moment that $l[i] == 1$.

```
Original code:

bool b = false;
while (i < n && !b){
    ...
    if (l[i] == 1)
        b = true;
    ...
}
```

```
Mutant code:

bool b = false;
while(i < n){
    ...
    if (l[i] == 1)
        b = true;
    ...
}
```

Preconditions: The *RCL* performance-mutant can be generated when there are other conditions that can stop the loop at some point. This happens when the controlling expression of the loop is not empty (cases 1 and 2) or when there is more than one condition in the controlling expression of the loop (case 3).

5.2. Unnecessary Recalculation of Values (URV)

Motivation: Olivo et al. [16] found some situations where a program repeatedly iterates over a data structure that has not been modified between successive traversals. For instance, they identified the so-called *ExtremeVal* performance bug, which appears when the maximum or minimum element of a collection is calculated multiple times even though the collection never changes [16].

Definition: With the aim of generalizing the type of bug described in the motivation, the *URV* operator forces the recalculation of values previously computed. In more detail, this operator seeks for variables defined to store the value returned by an invocation to a method. Then, the variable is

Prepared using stvrauth.cls
DOI: 10.1002/stvr

deleted and each reference to that variable is replaced by an invocation to the same method so that the computation performed by this method is repeated.

Example: The following fragment of code shows the deletion of the *min* variable and the replacement of the references to that variable by calls to the *minValue* function, forcing the recalculation of the minimum value of *v1* $2 \times$ n times.

```
Original code:                    Mutant code:
int min = minValue(v1);
for(int i=0; i<n; i++){           for(int i=0; i<n; i++){
    if(v2[i] > min)                   if(v2[i] > minValue(v1))
        v2[i] = min;                      v2[i] = minValue(v1);
}                                 }
```

Preconditions: The *URV* performance-mutant can be generated when (1) there is a variable that stores a computed value and it is referenced more than once (case of equivalence), and (2) the variable that stores the computed value and the variables involved in its computation do not change between the point of the computation and the references to the variable.

5.3. Move/Copy Statement into Loop (MSL)

Motivation: Xu et al. [1] observed situations where many temporary objects were generated in a loop simply to provide a simple service (e.g., to assign a value to the fields of other objects). Yan et al. [10] also found that *Never-Used and Rarely-Used Allocations* (that is, objects that are created but hardly referenced) appear surprisingly often.

Definition: Based on the above performance bugs, the *MSL* operator searches for parts of the code where new objects are generated before a loop statement. The operator then moves the object creation statement into the loop. In this way, the performance-mutant generates many temporary objects inside the loop instead of just one. Additionally, this operator moves conditional statements placed before loops' openings into the loops. This mutation can cause a significant degradation when the condition involves the execution of a costly operation that is repeated many times.

In some situations, *MSL* has to copy the object or statement instead of moving it in order to preserve the syntax of the language. For instance, the *if* condition cannot be moved inside a loop when the conditional statement is an *if/else* statement; in that case, the *if/else* statement is preserved but the *if* condition is copied inside the loop.

Example: This example represents the mutation of moving an object creation statement inside a loop. In the performance-mutant, a new instance *o1* is generated in each iteration, which results in the creation of many objects and the consequent waste of resources.

```
Original code:                    Mutant code:

Foo o1;                           for(int i = 0; i < n; i++){
for(int i = 0; i < n; i++){           Foo o1;
    v[i] = v[i] + o1.getField();      v[i] = v[i] + o1.getField();
}                                 }
```

Preconditions: The *MSL* performance-mutant can be generated when (1) the constructed object is not modified inside the loop, (2) none of the variables involved in the construction of the object changes inside the loop, (3) a conditional statement invokes at least one method or function, that is,

Prepared using stvrauth.cls
DOI: 10.1002/stvr

the condition does not perform trivial operations (equivalence case), and (4) none of the variables involved in the conditions of the *if* statement changes inside the loop.

5.4. Swap of Operands in Condition (SOC)

Motivation: In cases where we have a multiple condition, the condition that takes more time to be evaluated is normally placed in the last position. This can avoid the evaluation of the most time-consuming condition depending on the result of the first lighter conditions.
Definition: Considering the *Skippable function* bug pattern (see operator *RCL*) and reminding us of the traditional operator *COR* (Conditional Operator Replacement), the *SOC* operator forces the evaluation of a condition that might not be necessary depending on the result of other conditions. Thus, the *SOC* operator swaps the operands in a condition linked by a binary logical operator (&& and ||) so that the most time-consuming condition is evaluated regardless of the other conditions.
Example: This example represents the compulsory evaluation and execution of the most costly method within the multiple conditions. As it can be seen in the original program, the invocation to *costlyMethod* can be saved when the condition *a != 1* evaluates to true.

```
Original code:                          Mutant code:

if(a != 1 || costlyMethod()){           if(costlyMethod() || a != 1){
    ...                                     ...
}                                       }
```

Preconditions: The *SOC* performance-mutant can be generated when (1) the condition invokes at least one method or function, that is, the condition does not perform trivial operations (case of equivalence), and (2) the most costly condition does not depend on the lighter conditions, such as in Figure 2.

5.5. Simulation of Heavy-Weight Operation (HWO)

Motivation: The study by Ongkosit and Takada [28] defines a list of potentially blocking operations in Android applications, such as bitmap processing and the access to network, storage, and databases. Linares-Vásquez et al. [19] defined three mutation operators to inject a long delay in some specific situations in Android applications too, like the insertion of a delay in the GUI listener thread. In general, some operations are more time-consuming than others and they can be especially problematic when they are contained in third-party libraries. That is, they can lead to developers' workload misunderstandings or they can vary the performance of a program depending on the context or the updates [10].
Definition: The *HWO* operator injects a delay *t* right after each invocation to methods in third-party libraries and known heavy-weight operations (storage access, network connection...) so that workload issues can be revealed. By default, the value of the delay *t* could be correlated with the execution time of the original program.
Example: This example shows the insertion of a delay, in the form of sleep operation, just after a call to a costly method.

```
Original code:              Mutant code:

costlyMethod(a1, a2);       costlyMethod(a1, a2);
```


DOI: 10.1002/stvr

```
sleep_for(std::chrono::seconds(t));
```

Preconditions: No preconditions are required by the *HWO* operator.

5.6. *Creation of Short-lived Objects (CSO)*

Motivation: The idea behind the *View Holder* pattern is to reuse previously recycled items instead of generating new objects [15]. Similarly, the motivating example in Figure 1 in the study by Yan et al. [10] shows that in some situations we can reuse a single object across multiple calls to the same method.

Definition: Inspired by the works exposed in the motivation, the *CSO* operator forces the generation of a new object every time a method is called instead of reusing an already created object. In detail, this operator targets the methods that receive an object as a parameter, which is used and returned by the same method. Then, the operator generates a clone of such objects, producing new short-lived objects inside the method every time the method is invoked.

Additionally, some languages provide other mechanisms to avoid the repetition of tasks that should be performed only once. Therefore, this operator could be adapted to generate more performance-mutants depending on the language of the application.

Example: This example represents the redundant creation of an object instance, *f2*, when the existing object *f* could be used.

```
Original code:

Foo multiply(Foo f, int n){
    return f.getNumber() * n;
}
```

```
Mutant code:

Foo multiply(Foo f, int n){
    Foo f2(f);
    return f2.getNumber() * n;
}
```

Preconditions: No preconditions are required by the *CSO* operator.

5.7. *Memory Space Reservation (MSR)*

Motivation: Sometimes, the initial capacity of collections is not appropriate for the number of elements stored. Using small initial sizes can lead to frequent *resize* operations. In contrast, the pre-allocation of a collection at the maximal number of elements results in very low utilization.

Definition: Based on the *Never-Used and Rarely-Used Allocations* bug pattern [10] and on the causes of low utilization of collections [17], the *MSR* operator allocates less or more space for dynamic collections. This operator modifies a collection with dynamic allocation to shrink or expand the reservation space for elements to simulate both cases. The value of the size increase t should be configured in the latter case.

Example: The example represents the two ways of applying the *MSR* operator: the space allocation for vector *v* is reduced to 1 (first case), and it is increased by a factor of t, being $t > 1$ (second case).

```
Original code:

vector<Foo> v(INIT_SIZE);
```

```
Mutant code:

vector<Foo> v(1);
```

```
Original code:
```

```
Mutant code:
```


DOI: 10.1002/stvr

```
vector<Foo> v(INIT_SIZE);          vector<Foo> v(INIT_SIZE * t);
```

Preconditions: No preconditions are required by the *MSR* operator.

6. EVALUATION

In this section we describe the test subjects, the process followed to classify the mutants and the experimental procedure used to analyze the feasibility of applying PMT.

6.1. Subjects under test

We selected 3 open-source projects in C++ that are distributed with functional test suites. Namely:

- *TinyXML2* [45] (Txm): An efficient parser of XML files that can be integrated into other C++ programs to read and write XML documents. It contains 2,620 lines of code (*LOC*).
- *XmlRPC++* [46] (Rpc): An XML-RPC protocol implementation to incorporate client-server communication through HTTP support into C++ applications. *LOC*: 2,194.
- *Dolphin* [47] (Dph): A navigational file manager used by KDE desktop applications. We have focused on the class *KFileItemModel*, which allows loading items of a directory. *LOC*: 2,771.

These programs were compiled and executed on a server running the operative system *Ubuntu* 14.04 and 32GB of RAM memory.
For the evaluation of PMT with traditional mutation operators (**RQ2**), we selected the same subset of traditional operators that López et al. [23] proposed to optimize the code by reusing equivalent mutants:

- Arithmetic Operator Replacement (*AOR*).
- Relational Operator Replacement (*ROR*).
- Assignment Shortcut Replacement (*ASR*).

We used the mutation tool *MuCPP* [35] for C++ programs to apply these traditional operators. In the case of performance-mutants (**RQ3** and **RQ4**), they were inserted manually into the programs due to the challenging task of automating these operators. To do so, in a first step, and having in mind each of the performance operators, we reviewed the code of the case studies searching for all the potential mutation locations. In a second phase, we analyzed in depth the list of locations and discarded those that did not meet the preconditions associated with these operators. Then, we applied the operators manually in the remaining locations to create the performance-mutants. Finally, we compiled the mutants to ensure that there were no invalid ones due to errors in the manual injection of mutations.

For the evaluation of the relation between PMT and compiler optimizations (**RQ5**), we applied the technique known as *Trivial Compiler Equivalence* or *TCE* [27]. This technique allows the automated detection of two equivalent programs when, as a result of the optimizations performed by compilers, the binary files of both programs turn out to be identical. In our work, we use the compiler *gcc*, a reliable compiler for all standards-compliant programs, to compare the binary files of the original

Prepared using stvrauth.cls
DOI: 10.1002/stvr

program and the performance-mutants. The compiler invalidates a performance-mutant when there is no difference between its binary files and the ones of the original program.

6.2. Experimental setup

This evaluation seeks to analyze the feasibility of applying PMT using both traditional and performance mutation operators. As a proof of concept, we focused on the execution time as the measurable non-functional property to assess the generated mutants. The approach should extrapolate to other properties like memory by using proper profilers that help detect significant deviations in the memory usage when compared to the original program.

We reused the functional test suite developed for these programs and we adapted it to work as a performance test suite. To do that, we executed the original program 30 times and recorded the average execution time of each test case. These recorded times acted as test oracles in our experiments by adding new time-related assertions. In particular, we followed the steps described below:

1. First, we executed the functional test suite on each generated mutant. This allowed us to discard the mutants that were killed by the functional test suite and therefore were obviously not valid for PMT, since they did not preserve the semantics of the program. The remaining mutants, the ones used in our study, were labeled as potentially valid (see Section 3).

2. Then, we executed each test case on each potentially valid mutant, and recorded the execution time of each test case individually. This step was repeated 30 times to get the average time of each test case in each of these mutants.

3. Finally, we compared the mean time of each test case in the original program and each mutant. In this way, we could classify the mutants into killed and alive depending on whether they increased the execution time in any of the test cases.

In order to determine whether each mutant was killed or not we performed a hypothesis statistical test. Specifically, for each subject program and mutant, we stated a null and alternative hypothesis. The null hypothesis (H_0) states that there is not a statistically significant difference between the execution times of the original program and the mutant, while the alternative hypothesis (H_1) states that the execution time of the mutant is higher than the execution time of the original program and such difference is statistically significant. Specifically, a mutant is killed when the execution time of at least one test case is statistically higher compared with the execution time of that test case against the original program. Statistical tests provide a probability (named p-value) ranging in [0, 1]. We used three different widely used p-value thresholds, namely 0.01, 0.05 and 0.1 under which results are considered to be statistically significant and are sufficient to reject the null hypothesis. Since the execution times collected do not follow a normal distribution (according to Shapiro-Wilk normality test), we used the Mann-Whitney U test for the analysis. Finally, we performed a manual review of the mutants that remained alive with *p-value<0.1*, in order to know how many of them were valid and whether we could find meaningful mutants.

Interruptions in the execution of a mutant caused by timeouts are normally easy to detect using either functional or non-functional test suites. Also, we cannot know whether the mutation will

DOI: 10.1002/stvr

finally cause a change in the behavior. Therefore, we discarded those mutants causing a timeout in our evaluation.

7. RESULTS

In the following subsections, we report the results and how they answer the research questions:

7.1. RQ1: Injection of performance mutations

Table II shows the number of performance mutations that were injected in the subjects under test based on the performance mutation operators defined in Section 5. The results are divided by mutation operator and program. As it can be seen, all mutation operators could be applied in the conducted experiment, being *Dolphin* the program in which more mutations were injected. Additionally, all operators could be applied in two of these three programs (*CSO* and *MSR* did not lead to any mutants in *TinyXML2*). The mutation operators *URV*, *SOC* and *HWO* were the most prolific operators, while we did not find many situations to apply *CSO* and *MSL*.

Table II. Performance-mutants divided by program and performance mutation operator

Program	RCL	URV	MSL	SOC	HWO	CSO	MSR	Total
Txm	11	10	2	9	7	-	-	39
Rpc	10	25	1	23	13	1	2	75
Dph	9	41	4	34	12	11	16	127
Total	**30**	**76**	**7**	**66**	**32**	**12**	**18**	**241**

Regarding the application of the performance operators in these programs, we should remark the following particular cases:

- Operator *HWO*: *TinyXML2* reads and writes XML files using specific operations of a library to that end. *Dolphin* emits *Qt* signals[†] to intercommunicate objects. Finally, *XmlRPC++* makes use of libraries to operate with sockets for client-server communication. The invocation to these functions has been the target of *HWO*. We set a generic delay of 1 second for these applications.
- Operator *MSR*: *Dolphin* employs *Qt* dynamic containers and sometimes the program reserves memory for them through specific methods. Similarly, *XmlRPC++* reserves memory for strings. This operator focuses on those operations. We set the value $t = 2$ (i.e., we double the reserved size).
- Operator *CSO*: *Dolphin* and *XmlRPC++* contain some methods that use static variables to reuse information between calls to the same method. These locations are the target of this operator.

Answer to RQ1: *How feasible is to define domain-independent performance mutation operators?* We have been able to define 7 mutation operators based on reported performance bugs in the

[†] http://doc.qt.io/qt-5/signalsandslots.html

Prepared using stvrauth.cls
DOI: 10.1002/stvr

literature. Most of these operators require to preprocess the code to avoid the generation of some mutants that are known to alter the semantics of the program. Some other operators may require the configuration of some parameters prior to their application, as in the case of *HWO* and *MSR*. Examples of performance-mutants based on all these mutation operators have been found in the three real-world programs, especially in the case of *URV* and *SOC*. The research questions RQ3 and RQ4 will explore whether these mutants can actually be useful to improve performance tests.

7.2. RQ2: Analysis of traditional mutants

Table III presents the results of the execution of traditional mutants to evaluate the test suite from a non-functional perspective. The table shows the number of generated *mutants* and how many of them are *potentially valid* mutants for PMT, and how many of them are *killed* and *alive* using the different thresholds of degradation. Recall that we removed mutants that raised the timeout, so the sum of killed and alive is not the same as the number of potentially valid mutants. The table also reports the mutants found as *valid* when the mutants that remained *alive* with *p-value<0.1* were manually reviewed.

From 587 mutants generated in the three programs, 251 mutants passed the first step, that is, only 42.7% of the mutants survived the functional test suite execution. These results are graphically shown in Figure 3 divided by mutation operator. Taking into account that using traditional operators is costly, this low percentage is a detriment to the application of PMT using this set of operators.

Table III. Classification of mutants generated by the set of traditional operators (Mut.), potentially valid (Pot. valid), killed and alive mutants with *p-value<0.1*, *p-value<0.05* and *p-value<0.01*, and valid mutants in the set of alive mutants with *p-value<0.1* once these mutants were manually reviewed.

Prog.	Mut.	***Pot. valid***		***p-value<0.1***		***p-value<0.05***		***p-value<0.01***		
		#	%	Killed	Alive	Killed	Alive	Killed	Alive	Valid
Txm	173	51	29.5%	7	41	4	44	0	48	6
Rpc	231	112	48.5%	65	17	58	24	45	37	0
Dph	183	88	48.1%	78	5	77	6	60	23	0
Total	587	251	42.7%	150	63	139	74	105	108	6

As expected, the number of killed mutants decreases as the threshold increases. In total, from the 105 mutants killed considering the most lenient threshold (*p-value<0.1*), 45 of them appear as surviving mutants with the most stringent one (*p-value<0.01*). We reviewed the live mutants (threshold 0.1) to analyze these mutants qualitatively. That set of live mutants accounts for 25.1% of the potentially valid mutants (63 out of 251). Even though it is difficult to determine whether these mutants are valid or not, it is likely that the functional test suite lacks specific test cases or asserts that are able to reveal most of these live mutants (57 out of 63). Indeed, it is also likely that many of the 150 killed mutants could also be detected using a better functional test suite. It is also remarkable that 30 mutants were detected with a timeout in *XmlRPC++*, which probably occurs because the mutation impedes the client-server communication.

We only found a few mutants from the set of alive mutants with *p-value<0.1* that we believe that do not change the functional behavior. Figure 4 shows an example of one of those mutants generated by the mutation operator *ROR* in *TinyXML2*. As illustrated, the mutation causes the program to allocate memory even when the number of elements to be stored (*cap*) is equal to the memory currently allocated (*_allocated*).

Prepared using stvrauth.cls
DOI: 10.1002/stvr

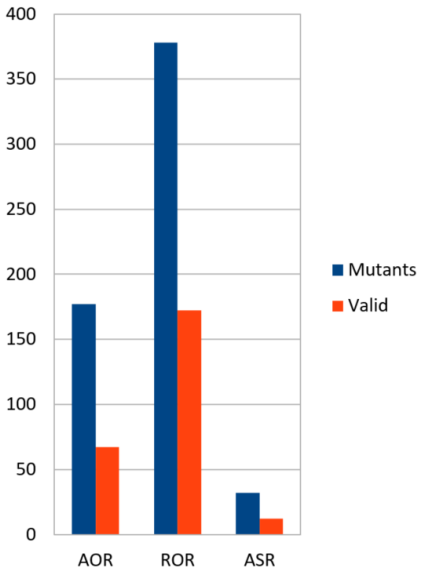

Figure 3. Number of mutants generated and potentially valid mutants by traditional mutation operator

```
1          void EnsureCapacity( int cap ) {
2      -     if ( cap > _allocated ) {
3      +     if ( cap /*ROR*/>= _allocated ) {
4              int newAllocated = cap * 2;
5              T* newMem = new T[newAllocated];
6              memcpy( newMem, _mem, sizeof(T)*_size );
7            ...
```

Figure 4. *ROR* mutant in *TinyXML2*

This situation is similar to the one identified by López et al. [23], where the change of relational operator produces that a skippable operation is executed.

Answer to RQ2: *What percentage of mutants generated by traditional operators can model performance bugs?* The three traditional operators selected (*AOR*, *ROR* and *ASR*) generate a large number of mutants, but only 42.7% of them are not detected by the functional test suite. Even though there is a considerable number of mutants that increased the execution time, a review of the live mutants reveals that few traditional mutations can really model meaningful performance bugs. As expected, these mutations appear in rare instances that are tied to the semantics of the program.

7.3. RQ3: Analysis of performance-mutants

Table IV shows the results of the execution of performance-mutants in a similar fashion to Table III.

At first sight, the most remarkable fact is that 80.5% of the generated performance-mutants passed the functional test suite (in contrast to 42.7% of the traditional mutants) . The number of mutants generated with these 7 performance operators is not so high as with the 3 traditional operators (241 vs 587 mutants), which is a positive aspect in mutation testing.

The tendency regarding killed mutants is similar to that of traditional mutants. The percentage of live traditional mutants is higher than the percentage of live performance-mutants except for

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Table IV. Classification of performance-mutants generated by the set of performance mutation operators (Mut.), potentially valid (Pot. valid), killed and alive mutants with *p-value<0.1*, *p-value<0.05* and *p-value<0.01*, and valid mutants in the set of alive mutants with *p-value<0.1* once these mutants were manually reviewed.

Prog.	Mut.	*Pot. valid*		*p-value<0.1*		*p-value<0.05*		*p-value<0.01*		
		#	%	Killed	Alive	Killed	Alive	Killed	Alive	Valid
Txm	39	24	61.5%	11	9	8	12	4	16	8
Rpc	75	56	74.6%	24	24	17	31	3	45	20
Dph	127	114	89.8%	105	4	101	8	68	41	4
Total	241	194	80.5%	140	37	126	51	75	102	32

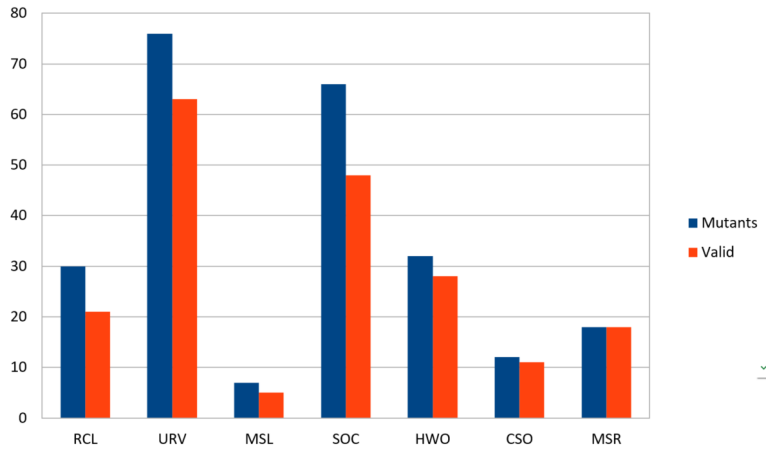

Figure 5. Number of mutants generated and potentially valid mutants by performance mutation operator

the threshold *p-value<0.01* (50.7% vs 57,6%). A review process reveals that 32 out of those 37 performance-mutants that remained alive with *p-value<0,1* do not change the functional behavior, and should be plausibly useful mutants to generate new test inputs. These numbers suggest that these performance-mutants are more difficult to detect than traditional ones, and may require special test scenarios beyond the ones typically designed to build functional test suites, which is in line with the observations in the literature (see Section 3). Only a few mutants change the functionality of the program. For example, a *SOC* mutation in *XmlRPC++* swaps two invocations to the method *nextTagIs*, which modifies the value of the input-output parameter *offset* (see Figure 6). The program expects to find the tag *PARAMS* before *PARAM* in the response (<*params*><*param*>). This mutation alters the semantics because the first call to *nextTagIs* updates the *offset* to the position of the tag *PARAM*, which makes impossible to find the tag *PARAMS* after that.

```
1          bool XmlRpcClient::parseResponse(XmlRpcValue& result)
2              int offset = 0;
3              ...
4          -   if ((XmlRpcUtil::nextTagIs(PARAMS_TAG,_response,&offset) &&
5                  XmlRpcUtil::nextTagIs(PARAM_TAG,_response,&offset) ...)
6          +    if ((XmlRpcUtil::nextTagIs(PARAM_TAG,_response,&offset) &&
7                  XmlRpcUtil::nextTagIs(PARAMS_TAG,_response,&offset)) ...)
8               ...
9          }
```

Figure 6. *SOC* mutant in *XmlRPC++*

DOI: 10.1002/stvr

Figure 5 presents a more detailed classification of the performance-mutants by mutation operator. The operator *MSR* was the only mutation operator where all the mutants generated are potentially valid for PMT, while *SOC* is the one that more frequently alters the semantics of the program.

Answer to RQ3: *How many performance-mutants do performance mutation operators generate and how often do these mutants change the functional behavior?* The number of performance-mutants generated by the 7 performance operators defined is less than half of the mutants produced by the 3 traditional operators (241 vs 587 mutants). In principle, over 80% of those performance-mutants do not change the functionality, which nearly doubles the percentage of potentially valid traditional mutants. A manual review of the performance-mutants that survived the execution of the non-functional test suite with *p-value<0.1* reveals that a high percentage of those mutants (86.5%) preserves the semantics of the program.

7.4. RQ4: Qualitative analysis of significant performance mutations

In the previous section, we manually reviewed the mutants that were alive with threshold *p-value<0.1*, and counted the number of them that could be classified as valid. In this section, we assess those mutants qualitatively in order to detect potentially helpful performance-mutants, even though we cannot say with certainty whether these mutants could be meaningful for testers of these programs.

There are several reasons why these mutants survived the test suite execution. Some of them simply require a test scenario that executes a fragment of code to reveal the performance issue. Some others can only be identified when executed on other platforms. For instance, *XmlRPC++* applies some operations with sockets which are specific for *Windows* systems. However, in general, most of these mutants would require special and large-scale inputs to be detected.

Below, we describe some of the reviewed mutants that can cause noticeable performance failures.

***URV* mutant:** The method shown in Figure 7 returns the value of an attribute with a given name in an XML element. If such an attribute does not exist or its value is different from the one expected, it returns 0. Since the method has to check for these conditions, the attribute is first searched by its name and saved in a reference variable *a* (line 3). The search by name is performed by the method *FindAttribute* and it requires to iterate the list of attributes and compare their names with the one specified. In this mutant, each reference to the variable *a* is replaced by a call to *FindAttribute*, which result in three searches instead of one (lines 6, 10 and 12).

Thanks to the above mutant, we found that a test case that processed an XML file with a long list of attributes in its elements was missing to detect this performance issue. This is a clear example of the kind of inputs that would be required to reveal these issues.

MSR mutant: The fragment shown in Figure 8 presents a list (*QList*) of objects *KUrl* (line 4). The size of the list is known in advance (*itemCount*) and the code reserves memory for that number of elements. The mutant disregards the prediction of the elements that will be appended (line 5), thereby causing a repetitive allocation of the data (*resize* operations) as the number of elements appended increases (line 8).

Prepared using stvrauth.cls
DOI: 10.1002/stvr

```
const char* XMLElement::Attribute( const char* name, const char* value ) const
{
-       const XMLAttribute* a = FindAttribute( name );
+        /* URV */ //const XMLAttribute* a = FindAttribute( name );
-       if ( !a ) {
+       if ( !FindAttribute( name ) ) {
            return 0;
        }
-       if ( !value || XMLUtil::StringEqual( a->Value(), value )) {
+       if ( !value || XMLUtil::StringEqual( FindAttribute( name )->Value(), value )) {
-           return a->Value();
+           return FindAttribute( name )->Value();
        }
        return 0;
}
```

Figure 7. *URV* mutant in *TinyXML2*

```
void KFileItemModel::resortAllItems(){
        ...
        const int itemCount = count();
        QList<KUrl> oldUrls;
-       oldUrls.reserve(itemCount);
+       /* MSR */ oldUrls.reserve(1);
        foreach (const ItemData* itemData, m_itemData) {
            oldUrls.append(itemData->item.url());
        }
        ...
}
```

Figure 8. *MSR* mutant in *Dolphin*

This mutant should lead to the design of a test case that calls the method *resortAllItems()* when there is a large number of items in *m_itemData*.

RCL mutant: The code fragment presented in Figure 9 shows an interesting mutant generated by the operator *RCL*. The code shows a loop that looks for a specific entity to print a pattern. Once the entity has been found, the *break* statement avoids useless iterations. The mutant removes that statement, which leads to a performance issue when the method is called many times or if there are many entities.

```
void XMLPrinter::PrintString( const char* p, bool restricted ){
        ...
        for( int i=0; i<NUM_ENTITIES; ++i ) {
            if ( entities[i].value == *q ) {
                Print( "&%s;", entities[i].pattern );
-               break;
+               /* RCL */
            }
        }
        ...
}
```

Figure 9. *RCL* mutant in *TinyXML2*

SOC mutant: The method *hasMember* shown in Figure 10 searches for a member with a given name provided that the object is of type *TypeStruct*. The mutation swaps the check of the two

Prepared using stvrauth.cls
DOI: 10.1002/stvr

conditions (name and type). This mutant forces the method to search for the member in a *map* container regardless of the type, thus increasing the execution time.

```
1   bool XmlRpcValue::hasMember(const std::string& name) const
2   {
3   -    return _type == TypeStruct && _value.asStruct->find(name) != _value.asStruct->end();
4   +    /* SOC */ return _value.asStruct->find(name) != _value.asStruct->end() && _type == TypeStruct;
5   }
```

Figure 10. *SOC* mutant in *XmlRPC++*

This mutant should lead to the design of a test case that invokes this method several times from a *XmlRpcValue* object with *_type* different to *TypeStruct* and with the *map* container previously filled with several members.

Answer to RQ4: *Are the defined performance operators able to model some noticeable performance faults?* Yes. Even though we have found some performance-mutants that seem trivial to kill, we have also found some exemplary mutants extracted from the subjects under test that illustrate the kind of performance issues caused by the new mutation operators, and we have proposed new test scenarios to improve the current test suites.

7.5. RQ5: Compiler optimizations

Table V shows the results of the application of Trivial Compiler Equivalence (TCE) to performance-mutants. We may recall that the objective was to check whether the optimizations performed by the compiler undo some of the changes introduced by the performance mutation operators. In particular, we compiled with no optimization settings and with the highest optimization level supported by the *gcc* compiler (*-O3*). As it can be seen, the results are similar in both cases: 8 mutants are reverted with the default option altogether in the three programs, and 10 mutants with level *-O3*. We have to note that the two performance-mutants reverted in *TinyXML2* and all but the *CSO* mutant in *XmlRPC++* corresponded to the same circumstance: the mutation was injected into a fragment of code that is activated in *Windows* systems through a preprocessor directive. Therefore, they were reverted by the compiler because we executed the experiments on a server running *Ubuntu*.

As a result, there are only two performance-mutants (generated by *URV* and *SOC*) invalidated by the compiler when applying the optimization level *-O3*. Figure 11 shows the reverted mutant generated by *SOC*, which swaps two similar conditions.

Table V. Number of mutants reverted by the compiler applying no optimization settings (*None*) and the highest optimization level (*-O3*). The results are divided by mutation operator and program under test.

Operator	***Txm***		***Rpc***		***Dph***	
	None	-O3	None	-O3	None	-O3
RCL	-	-	-	-	-	-
URV	-	-	-	**1**	-	-
MSL	-	-	-	-	-	-
SOC	-	-	-	-	-	**1**
HWO	2	2	5	5	-	-
MSR	-	-	-	-	-	-
CSO	-	-	1	1	-	-
Total	2	2	6	7	0	1

DOI: 10.1002/stvr

```
1       for (int i = 0; i <= maxIndex; ++i) {
2           ...
3         - } else if (newFirstChar >= QLatin1Char('0') && newFirstChar <= QLatin1Char('9')) {
4         + } else if (newFirstChar <= QLatin1Char('9') && newFirstChar >= QLatin1Char('0')) {
5                 ...
6           }
7           ...
8       }
```

Figure 11. *SOC* mutant in *Dolphin*

Answer to RQ5: *Can optimizations performed by compilers revert the mutations injected with performance operators?* Only a couple of performance-mutants generated by the operators *SOC* and *URV* are invalidated applying TCE. These two mutants are only reverted by the compiler when an optimization level is set. There are also some other mutations injected by the operators *HWO* and *CSO* that are not useful in our experiments because we use *Ubuntu* and these mutations can only be activated in *Windows* systems through preprocessor directives.

7.6. Additional remarks

The number of killed mutants was low in the experiments compared to traditional mutation, especially as the threshold increased. The fact that we adapted test suites focused on the functional aspect to work as performance test suites has probably prevented a higher number of performance-mutants from being initially detected. An interesting aspect regarding equivalence in PMT is that the number of equivalent performance-mutants depends on the threshold. A coarse adjustment of the threshold could lead to many potentially valid mutants. However, with a fine-grained threshold, equivalent performance-mutants should not be frequent, which is a positive aspect in mutation testing. This factor also depends on the accuracy of the methods to measure the assessed performance properties. Additionally, and following the RIP model proposed by Ammann and Offutt [48], once the program has been infected (which in PMT implies a noticeable increase in the measured non-functional properties), the cases where that infection does not propagate to the output should be rare. In other words, it is unlikely that the effects of the infection are masked after the execution of the mutated statement, which is a possible cause for the appearance of equivalent mutants in traditional mutation testing.

Several papers in the past have evaluated the significance of mutation testing when applied in real scenarios [49, 50]. These studies have been mostly performed using known traditional operators. Under the assumption that the hypotheses behind traditional mutation testing hold for PMT (see Section 3), we could expect that the benefits of applying the mutation-based approach hold for this new domain. However, we are still distant from being able to support this claim. First, we need to assess the level of automation that can be reached; then, only experiments with real case studies could give evidence of the ability of mutation-adequate performance tests to find performance bugs.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

8. THREATS TO VALIDITY

Internal validity:

The injection of performance mutations in the experimental evaluation has been done in a manual way, which is error-prone. We have proceeded with care to generate all possible performance-mutants taking into account the constraints of each operator. Despite the manual injection of mutants, we have followed a systematic process consisting of four steps in the hope that it can be reproduced by any testers. Also, live performance-mutants have been reviewed manually to determine whether they could model interesting performance issues, being this process also prone to errors. We could have generated new test cases to kill performance-mutants. However, that would require both a deep knowledge of the program under test and the generation of large-scale inputs, which are not easy to design. There is the possibility that some of the mutants that survived the test suite could be detected with new functional test cases, but this would require the design of an adequate test suite, which would be a hard and time-consuming task.

Construct validity:

As mentioned earlier, the output of the executions in terms of time consumption is not deterministic and can be affected by the current computer workload or the specific characteristics of the machine. To mitigate this threat, we have compared 30 executions of each test case on the original program and each mutant.Additionally, we have performed a statistical analysis of the execution times and we have used three different thresholds (p-values). More complex methods could be used, like the ones proposed by Reichelt and Kühne [9]. However, the methods that require many executions would make the application of the technique excessively expensive.

External validity:

The aim of our evaluation was to assess the feasibility of the definition and application of domain-independent performance operators, as this is the first evaluation of such operators at the source-code level as far as we know. As a proof of concept, we have used and analyzed three test subjects coded in C++ language. Further experiments are required to know whether these results hold in other programs and programming languages. Also, we have used a subset of traditional operators in the evaluation, but there is the possibility that other traditional operators could be useful for PMT. Nevertheless, we have focused on three widely-used traditional operators that, additionally, have been proposed to optimize the code in terms of performance in a related work [23]. We have applied *gcc* as the compiler to observe whether performance mutations could be reverted by compiler optimizations, as this is the more widely-used compiler for C++ programs. However, new experiments should be conducted with compilers for other programming languages.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

9. RELATED WORK

9.1. Specification-based mutation operators

Mutation operators to test for timely behavior have been defined by Nilsson et al. [20] and AbouTrab et al. [51]. These operators mutate time constraints set in timed automata, where most of them focus on modifying the time boundaries of these constraints (i.e., increase or reduce each time property by a *delta*). Later, Lindstrom et al. [33] suggested to set the *delta* individually for each mutant generated with the operators defined by Nilsson et al. [20]; depending on the situation, a high *delta* would lead to the generation of trivial mutants (i.e., detected with simple test cases) and a low *delta* would lead to the creation of many equivalent mutants. Temple et al. [21] proposed the generation of variants of the program, called *morphs*, that modify the properties of software product lines and that can result in performance issues. Finally, Vega et al. [22] have recently performed a review focused on model-based mutation testing for timed systems.

9.2. Domain-specific mutation operators

A first strategy used to address the detection of performance bugs is the insertion of delays to test the responsiveness of applications [30]. However, the most common strategy to detect and fix bugs is to study real cases of bugs found in existing projects. This analysis can reveal patterns in the code that usually lead to this kind of bugs (i.e., bug patterns). Such bug patterns can then be automatically detected by means of rule-based detection using static analysis. Bug patterns related to the functional behavior have been collected by Pan et al. [14]. Similarly, bug patterns related to the performance behavior have been proposed by Jin et al. [7], Liu et al. [15], and Ongkosit and Takada [28] for Android applications. Recently, Linares-Vásquez et al. [19] defined a complete taxonomy of Android-related and Java-related performance bugs, which includes a subcategory of non-functional requirements.

All these studies have something in common: either they are focused on a specific language or context (such as Android [15] or desktop GUI applications [11]) or they describe specific bug patterns. For instance, Jin et al. [7] found that the function `random` should not be used by concurrent threads and that the function `aggregateTransact` should replace the invocations to `doTransact` when they are inside loops. While detecting and fixing these situations in the code can avoid specific performance issues, focusing only on these concrete cases is not effective in performance testing because we may miss some other kind of bugs that cause significant degradation. Additionally, the findings in that study are not applicable to other languages since those rules focus on specific functions only available in Java.

9.3. Traditional mutation operators to address non-functional properties

Program optimization to identify and fix some known performance issues is the target of a large body of research, from the diagnosis of energy inefficiency [2] to the improvement of the memory footprint [17]. Mutations have also been applied for optimization purposes. Langdon et al. [44] focused on software refactoring through genetic programming to optimize the performance of a program. Recently, López et al. [23] proposed using classical mutation operators applicable to different programming languages to find mutants with better non-functional aspects. However, the

DOI: 10.1002/stvr

experiments in this paper show that it is unlikely that we can aspire to have a lot of useful mutants if we keep applying the same mutation operators that we have used so far to mimic behavioral faults. Indeed, the ratio of effective mutants may decrease as we add more traditional mutation operators, which is a problem taking into account the high cost of applying mutation testing. As a result of this issue, Sánchez et al. [26] suggested the use of an evolutionary algorithm to guide the search towards useful traditional mutants for the refinement of performance tests.

A recent research work presented by Lisper et al. [24] introduced the concept of targeted mutation, which aims at non-functional properties. This approach outlines the idea of introducing mutants to guide the generation and augmentation of test suites to be used for estimating the worst-case execution time. In contrast, our work pursues the definition and application of specific mutation operators affecting non-functional properties, such as the memory consumption or execution time among others.

There are some other works related to extra-functional properties that generate mutants with the sole aim of altering the system's functionality. For instance, Aichernig et al. [41] also focused on timed automata but changing different aspects of the syntax of the model instead of their time properties. Also, Wu et al. [52] proposed a set of mutation operators related to memory with the classical goal of perturbing the functional behavior of the program.

10. CONCLUSION AND FUTURE WORK

In this work, we have investigated the possibility to extend the current state of the art regarding mutation testing to evaluate and improve performance tests. The main finding of our study is that classical methods used in functional testing are not really effective when it comes to assess and improve non-functional tests, and that specific mutation operators and mechanisms to evaluate the outputs are necessary to successfully apply performance mutation testing. In particular, we have defined and evaluated seven new performance mutation operators that have highlighted the need for new performance tests. However, an analysis of a larger set of traditional operators would be desirable to confirm this finding, including those operators that usually produce a high number of equivalent mutants.

As far as we know, there are not many works that have tackled this research line, so much work lies ahead regarding the possible definition of new performance operators and the selection of the most effective ones. Also, its real effectiveness in practice and in different domains, or the best methods to detect performance issues associated with different non-functional properties should be further explored. Another challenge for future work is the automation of these performance operators. Static analysis techniques, used in other mutation tools like MuCPP [35], could be useful in a basic version of these operators. In a more advanced version, a data-flow analysis technique could be required to apply the preconditions required for some of these operators. LLVM's analysis passes [53] could be helpful to collect important information before applying the mutations. However, it is still unknown how costly such a method would be and whether it would compensate the number of mutants that would be actually avoided in real scenarios. Another point to consider is that it is unlikely that the effects of a mutation in PMT are masked, that is, most mutations should

DOI: 10.1002/stvr

propagate to the output. As such, it is reasonable to think that the combination of mutations in high-order mutation could be more effective than first-order mutation. New experiments that compare both techniques could shed light about this issue in the future.

11. ACKNOWLEDGEMENTS

This work has been partially supported by the European Commission (FEDER), Spanish Government under MINECO projects DArDOS (TIN2015-65845-C3-3-R), FAME (RTI2018-093608-B-C33), BELI (TIN2015-70560-R) and HORATIO (RTI2018-101204-B-C21) and the Operational Programme Andalusia under project APOLO (US-1264651).

REFERENCES

1. Xu G, Arnold M, Mitchell N, Rountev A, Sevitsky G. Go with the flow: Profiling copies to find runtime bloat. *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, ACM: New York, NY, USA, 2009; 419–430, doi:10.1145/1542476.1542523. URL http://doi.org/10.1145/1542476.1542523.
2. Liu Y, Xu C, Cheung SC, Lu J. Greendroid: Automated diagnosis of energy inefficiency for smartphone applications. *IEEE Transactions on Software Engineering* 2014; (1):1–1.
3. Nistor A, Jiang T, Tan L. Discovering, reporting, and fixing performance bugs. *Working Conference on Mining Software Repositories*, 2013; 237–246.
4. Zaman S, Adams B, Hassan AE. A qualitative study on performance bugs. *IEEE Working Conference on Mining Software Repositories*, 2012; 199–208.
5. URL https://econsultancy.com/site-speed-case-studies-tips-and-tools-for\-improving-your-conversion-rate/, accessed: 14/10/19.
6. URL https://blog.kissmetrics.com/wp-content/uploads/2011/04/loading-time.pdf, accessed: 14/10/19.
7. Jin G, Song L, Shi X, Scherpelz J, Lu S. Understanding and detecting real-world performance bugs. *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, ACM: New York, NY, USA, 2012; 77–88, doi:10.1145/2254064.2254075. URL http://doi.org/10.1145/2254064.2254075.
8. Molyneaux I. *The Art of Application Performance Testing: Help for Programmers and Quality Assurance*. O'Reilly Media, 2009.
9. Reichelt DG, Kühne S. How to detect performance changes in software history: Performance analysis of software system versions. *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, ACM: New York, NY, USA, 2018; 183–188, doi:10.1145/3185768.3186404. URL http://doi.org/10.1145/3185768.3186404.
10. Yan D, Xu G, Rountev A. Uncovering performance problems in Java applications with reference propagation profiling. *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, IEEE Press: Piscataway, NJ, USA, 2012; 134–144. URL http://dl.acm.org/citation.cfm?id=2337223.2337240.
11. Jovic M, Hauswirth M. Listener latency profiling: Measuring the perceptible performance of interactive Java applications. *Science of Computer Programming* 2011; **76**(11):1054 – 1072, doi:10.1016/j.scico.2010.04.009. URL https://doi.org/10.1016/j.scico.2010.04.009, special Issue on Principles and Practice of Programming in Java (PPPJ 2008).
12. Vokolos FI, Weyuker EJ. Performance testing of software systems. *Proceedings of the 1st International Workshop on Software and Performance*, ACM, 1998; 80–87.
13. Segura S, Troya J, Durán A, Ruiz-Cortés A. Performance metamorphic testing: Motivation and challenges. *International Conference on Software Engineering: New Ideas and Emerging Results Track*, 2017; 7–10.

DOI: 10.1002/stvr

14. Pan K, Kim S, Whitehead EJ. Toward an understanding of bug fix patterns. *Empirical Software Engineering* Jun 2009; **14**(3):286–315, doi:10.1007/s10664-008-9077-5. URL https://doi.org/10.1007/s10664-008-9077-5.
15. Liu Y, Xu C, Cheung SC. Characterizing and detecting performance bugs for smartphone applications. *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, ACM: New York, NY, USA, 2014; 1013–1024, doi:10.1145/2568225.2568229. URL http://doi.org/10.1145/2568225.2568229.
16. Olivo O, Dillig I, Lin C. Static detection of asymptotic performance bugs in collection traveresals. *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015; 369–378.
17. Shacham O, Vechev M, Yahav E. Chameleon: adaptive selection of collections. *ACM Sigplan Notices*, vol. 44, ACM, 2009; 408–418.
18. Papadakis M, Kintis M, Zhang J, Le Traon Y, Harman M. Mutation testing advances: An analysis and survey. *Advances in Computers* 2017; .
19. Linares-Vásquez M, Bavota G, Tufano M, Moran K, Di Penta M, Vendome C, Bernal-Cárdenas C, Poshyvanyk D. Enabling mutation testing for Android apps. *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, ACM: New York, NY, USA, 2017; 233–244, doi:10.1145/3106237.3106275. URL http://doi.org/10.1145/3106237.3106275.
20. Nilsson R, Offutt J, Mellin J. Test case generation for mutation-based testing of timeliness. *Electronic Notes in Theoretical Computer Science* 2006; **164**(4):97 – 114, doi:10.1016/j.entcs.2006.10.010. URL https://doi.org/10.1016/j.entcs.2006.10.010, proceedings of the Second Workshop on Model Based Testing (MBT 2006).
21. Temple P, Acher M, Jézéquel JM. Multimorphic testing. *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ICSE '18, ACM: New York, NY, USA, 2018; 432–433, doi:10.1145/3183440.3195043. URL http://doi.org/10.1145/3183440.3195043.
22. Vega JJO, Perrouin G, Amrani M, Schobbens P. Model-based mutation operators for timed systems: A taxonomy and research agenda. *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, 2018; 325–332, doi:10.1109/QRS.2018.00045. URL https://doi.org/10.1109/QRS.2018.00045.
23. López J, Kushik N, Yevtushenko N. Source code optimization using equivalent mutants. *Information and Software Technology* 2018; **103**:138 – 141, doi:10.1016/j.infsof.2018.06.013. URL https://doi.org/10.1016/j.infsof.2018.06.013.
24. Lisper B, Lindström B, Potena P, Saadatmand M, Bohlin M. Targeted mutation: Efficient mutation analysis for testing non-functional properties. *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2017; 65–68, doi:10.1109/ICSTW.2017.18.
25. Sánchez AB, Delgado-Pérez P, Segura S, Medina-Bulo I. Performance mutation testing: Hypothesis and open questions. *Information and Software Technology* 2018; **103**:159 – 161, doi:https://doi.org/10.1016/j.infsof.2018.06.015. URL 10.1016/j.infsof.2018.06.015.
26. Sánchez AB, Delgado-Pérez P, Medina-Bulo I, Segura S. Search-based mutation testing to improve performance tests. *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '18, ACM: New York, NY, USA, 2018; 316–317, doi:10.1145/3205651.3205670. URL http://doi.org/10.1145/3205651.3205670.
27. Papadakis M, Jia Y, Harman M, Le Traon Y. Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE'15, IEEE Press: Piscataway, NJ, USA, 2015; 936–946, doi:10.1109/ICSE.2015.103. URL http://dx.doi.org/10.1109/ICSE.2015.103.
28. Ongkosit T, Takada S. Responsiveness analysis tool for Android application. *Proceedings of the 2Nd International Workshop on Software Development Lifecycle for Mobile*, DeMobile 2014, ACM: New York, NY, USA, 2014; 1–4, doi:10.1145/2661694.2661695. URL http://doi.org/10.1145/2661694.2661695.
29. Nistor A, Chang P, Radoi C, Lu S. Caramel: Detecting and fixing performance problems that have non-intrusive fixes. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, 2015; 902–912, doi:10.1109/ICSE.2015.100.
30. Yang S, Yan D, Rountev A. Testing for poor responsiveness in android applications. *2013 1st International Workshop on the Engineering of Mobile-Enabled Systems (MOBS)*, 2013; 1–6, doi:10.1109/MOBS.2013.6614215.
31. Han S, Dang Y, Ge S, Zhang D, Xie T. Performance debugging in the large via mining millions of stack traces. *2012 34th International Conference on Software Engineering (ICSE)*, 2012; 145–155, doi:10.1109/ICSE.2012.6227198.
32. Zhang P, Elbaum S, Dwyer MB. Automatic generation of load tests. *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, IEEE Computer Society: Washington, DC, USA, 2011; 43–52, doi:10.1109/ASE.2011.6100093. URL http://dx.doi.org/10.1109/ASE.2011.6100093.

33. Lindström B, Offutt J, Gonzalez-Hernandez L, Andler SF. Identifying useful mutants to test time properties. *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2018; 69–76, doi:10.1109/ICSTW.2018.00030.
34. Ma YS, Offutt J, Kwon YR. MuJava: an automated class mutation system. *Software Testing, Verification and Reliability* 2005; **15**(2):97–133.
35. Delgado-Pérez P, Medina-Bulo I, Palomo-Lozano F, García-Domínguez A, Domínguez-Jiménez JJ. Assessment of class mutation operators for C++ with the MuCPP mutation system. *Information and Software Technology* 2017; **81**:169 – 184, doi:10.1016/j.infsof.2016.07.002. URL https://dx.doi.org/10.1016/j.infsof.2016.07.002.
36. Derezińska A. Quality assessment of mutation operators dedicated for C# programs. *2006 Sixth International Conference on Quality Software (QSIC'06)*, 2006; 227–234, doi:10.1109/QSIC.2006.51. URL https://dx.doi.org/10.1109/QSIC.2006.51.
37. Shahriar H, Zulkernine M. Mutec: Mutation-based testing of cross site scripting. *Proceedings of the 2009 ICSE Workshop on Software Engineering for Secure Systems*, IEEE Computer Society, 2009; 47–53.
38. Delamaro ME, Maldonado JC, Mathur AP. Interface mutation: An approach for integration testing. *IEEE Transactions on Software Engineering* May 2001; **27**(3):228–247.
39. Lee SC, Offutt AJ. Generating test cases for XML-based web component interactions using mutation analysis. *Proceedings of the 12th International Symposium on Software Reliability Engineering (ISSRE'01)*, Hong Kong, China, 2001; 200–209.
40. Fabbri SPF, Delamaro ME, Maldonado JC, Masiero P. Mutation analysis testing for finite state machines. *Proceedings of the 5th International Symposium on Software Reliability Engineering*, Monterey, California, 1994; 220–229.
41. Aichernig BK, Lorber F, Ničković D. Time for mutants — model-based mutation testing with timed automata. *Tests and Proofs*, Veanes M, Viganò L (eds.), Springer Berlin Heidelberg: Berlin, Heidelberg, 2013; 20–38.
42. Offutt AJ. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering Methodology* Jan 1992; **1**(1):5–20, doi:10.1145/125489.125473. URL http://doi.org/10.1145/125489.125473.
43. Gopinath R. On the Limits of Mutation Analysis. Dissertation, Oregon State University jun 2017. URL http://hdl.handle.net/1957/61528.
44. Langdon WB, Harman M. Optimizing existing software with genetic programming. *IEEE Transactions on Evolutionary Computation* 2015; **19**(1):118–135.
45. TinyXML2. https://github.com/leethomason/tinyxml2. Accessed: 14/10/19.
46. XmlRPC++. http://xmlrpcpp.sourceforge.net/. Accessed: 14/10/19.
47. Dolphin. https://www.kde.org/applications/system/dolphin. Accessed: 14/10/19.
48. Ammann P, Offutt J. *Introduction to Software Testing*. 1st edn., Cambridge University Press: New York, NY, USA, 2008.
49. Daran M, Thévenod-Fosse P. Software error analysis: A real case study involving real faults and mutations. *SIGSOFT Software Engineering Notes* May 1996; **21**(3):158–171, doi:10.1145/226295.226313. URL http://dx.doi.org/10.1145/226295.226313.
50. Delgado-Pérez P, Habli I, Gregory S, Alexander R, Clark J, Medina-Bulo I. Evaluation of mutation testing in a nuclear industry case study. *IEEE Transactions on Reliability* Dec 2018; **67**(4):1406–1419, doi:10.1109/TR.2018.2864678.
51. AbouTrab MS, Brockway M, Counsell S, Hierons RM. Testing real-time embedded systems using timed automata based approaches. *Journal of Systems and Software* 2013; **86**(5):1209 – 1223, doi:10.1016/j.jss.2012.12.030. URL https://doi.org/10.1016/j.jss.2012.12.030.
52. Wu F, Nanavati J, Harman M, Jia Y, Krinke J. Memory mutation testing. *Information and Software Technology* 2017; **81**:97–111, doi:10.1016/j.infsof.2016.03.002. URL https://doi.org/10.1016/j.infsof.2016.03.002.
53. LLVM's passes. https://llvm.org/docs/Passes.html. Accessed: 14/10/19.

DOI: 10.1002/stvr