Check for updates

SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2021;e1769
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/stvr.1769

Analysing the combination of cost reduction techniques in Android mutation testing

Macario Polo-Usaola[1,*,†] and Isyed Rodríguez-Trujillo[2]

[1] *Departamento de Tecnologías y Sistemas de Información, University of Castilla-La Mancha, Ciudad Real, Spain*
[2] *Departamento de Computación, University of Concepción, Concepción, Chile*

SUMMARY

When applied to mobile software, mutation testing is particularly costly due to the deployment of the app under test onto the device: if one deployment is made for each generated mutant, the execution time becomes unapproachable. This paper analyses how the combination of different cost reduction techniques improves the execution time of mutation testing in mobile apps. The techniques reviewed and combined are *mutant schema*, *parallel execution* and two different ways of executing tests against the mutants (*all against all* and *all against mutants remaining alive*), as well as greedy algorithm for reducing the test suite size. This paper also presents a mathematical model of cost reduction and checks its validity with several experiments. Furthermore, the exhaustive and long experimentation has led the authors to compile a set of good practices which are also presented in a set of lessons learned. © 2021 John Wiley & Sons, Ltd

Received 14 February 2019; Revised 8 February 2021; Accepted 17 February 2021

KEY WORDS: cost prediction; mobile testing; mutant schema; mutation testing; parallel execution

1. INTRODUCTION

Mutation testing builds on discovering the artificial faults inserted in copies of the System Under Test (SUT) which are called mutants. The effectiveness of mutation testing has been demonstrated in many empirical studies [1] although it has the important drawback of its high computational cost, which is closely related to the number of mutants generated. Thus, cost reduction in mutation testing is a very active research matter [2–3]. Most of these studies are focused on the reduction of the number of mutants and on how tests are executed against mutants.

The situation is especially hard in the testing of mobile software. As Deng *et al*. point out [4], ‘for a variety of technical reasons, test execution [in Android] tends to be quite slow’, which ‘is particularly troublesome for Android testers’. These authors report that ‘a single iteration of an experiment required more than 20 hours’. In the experimentation carried out in this article, we have often far exceeded that value.

This is due to the nature of *instrumented* Android test cases. In Android, test suites can be composed of ‘unit’ or ‘instrumented’ test cases:

- *Unit* test cases can be executed as classical JUnit tests, directly on the developer’s computer, and their execution is relatively fast.
- *Instrumented* test cases simulate interactions of the user with the application or use Android-specific resources (sensors, for example). These tests require the application under test to be deployed and installed on an emulator or mobile device.

*Correspondence to: Macario Polo-Usaola, Departamento de Tecnologías y Sistemas de Información, University of Castilla-La Mancha, Ciudad Real, Spain.
†E-mail: macario.polo@uclm.es

© 2021 John Wiley & Sons, Ltd

Instrumented tests turn compilation, deployment, and installation into highly costly tasks, slowing down the whole mutation process. This is also highlighted by Escobar-Velásquez *et al.* [5], for whom 'Time is an issue in mutation testing, for both generation and testing time'.

This paper analyses the influence of several well-known cost reduction techniques on the mutation testing cost of Android applications. So, although this is a work about mutation testing, its goal is neither to validate nor to propose mutation operators, and neither to propose techniques to identify equivalent mutants. Our goal is to provide some kind of model to guide research and development of cost reduction techniques in mutation testing. Ideally, the proposed models will allow the theoretical study of cost reduction techniques to be undertaken before all the necessary infrastructures are developed and before starting the very costly experimentation phase.

This article describes a mathematical model that can be used to estimate a priori the time required to perform a mutation test on a mobile app. The model describes the expected execution time considering the use of Mutant Schema, Parallel execution and two different test case execution algorithms. The fit of the mathematical model is validated with a set of mobile apps, and it could be extended to consider other cost-reduction techniques. Although the model is applicable to any other context, our motivation for developing it has been mobile-software testing because of its high cost.

As shown in Section 3, several researchers have applied mutation testing to mobile software. To our best knowledge, all of them are focused on the proposal and development of mutation operators for this specific context, but none deals with the problem of execution time. Nonetheless, they all mention it as one of the biggest obstacles in mobile mutation testing.

Besides the mathematical model and the analysis of combinations of cost reduction techniques, the article presents a set of lessons learned (Section 7) and proposes some future lines of work (Section 8). Regarding the former, for instance, we redefine classic mutation processes [6, 7], now introducing the expected test-case execution time as an influencing factor to prioritize test-cases execution. Related to the latter, we point out some lines of research that undoubtedly will contribute to improve and to extend mutation testing not only to mobile software but also to other kinds of systems.

We also give a brief description of *BacterioWeb*, a tool we have developed for supporting the whole mutation testing process of mobile software through the web.

This paper is structured as follows: Section 2 overviews how Android apps are compiled, installed and tested, which is required for understanding the root cause of our problem. Section 3 is divided into two subsections, the first one describes mutation testing on mobile apps, while the second one reviews some cost reduction techniques in mutation testing. Section 4 defines several mathematical models of the cost reduction techniques applied. Section 5 describes the Mutant Schema approach used. Section 6 describes the experimental validation. Section 7 presents some lessons learned as a result of experimentation. Section 8 draws some possible future lines of work. We finish the paper with some conclusions in Section 9.

2. STRUCTURE AND TEST OF ANDROID APPLICATIONS

Android apps are usually written in Java or Kotlin and packaged for installation on the device as an *.apk* file. When compiled, the *.java* or *.kt* source files are translated into their corresponding *.class* files (made up of a Java Virtual Machine compatible bytecode), and from there, a second stage of compilation translates them into *.dex* files. Together with the resource files, the *.dex* files are packaged into an *.apk* file which contains the app. On the physical device, *dex* files are interpreted either by the *Dalvik* or by the ART (*Android Runtime*) virtual machines, depending on the Android version. Therefore, there are two compilation steps (from *.java* to *class* and from *.class* to *dex*) and one packaging step before the application is installed on the device.

Figure 1 illustrates a supposed app (*app.apk)* composed of three classes (*Screen1* and *Screen2*, which conform the user interface and are specializations of *Activity*, and *DomainObject*, which does not have any relation with the Android libraries). In the example, we have two different types of tests:

© 2021 John Wiley & Sons, Ltd

DOI: 10.1002/stv

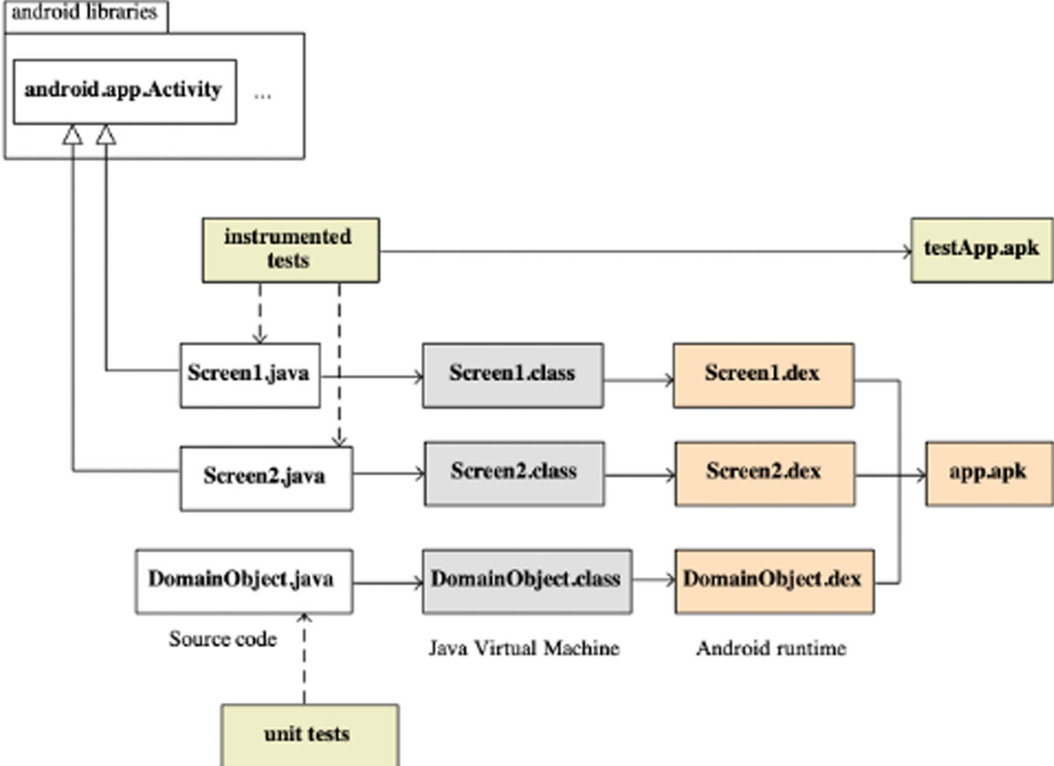

Figure 1. Compilation and packaging of an Android app.

- *Unit tests*, which exercise the business logic. These test cases do not need any special Android resource. These test suites do not need the construction of an *apk* and can be executed without any device.
- *Instrumented tests*, whose test cases require special Android resources for interacting with the user interface (clicking, writing …), using sensors and so on. Its execution requires the production of a separated *apk* file (*testApk.apk* in the example), which must be installed on the device.

Although the *testApp.apk* file only needs one deployment, a classic mutation approach [6, 7] requires that a different version of the *app.apk* is compiled, packaged and installed on the device for each mutant. The costs of compiling, packaging and installing are so high that testing a mobile app with a classic mutation testing process becomes almost completely impracticable. Thus, mobile software testing is an especially suitable context for applying cost reduction techniques.

Figure 2 shows part of the actual organization of the test files in *WordPress*, which is one of the apps we have used in our experiments. Instrumented and unit tests are respectively located under the *androidTest* and *test* folders. Files in these folders can be auxiliary classes (mocks, for example) or test suites. When the tests are to be executed from the official IDE (Android Studio), this one:

- For the instrumented test cases, the IDE creates a *testApp.apk* and one *app.apk* file. Both apks are pushed and installed on the mobile device (either an emulator or a physical device). Then, the IDE opens a virtual terminal on the device (whose operating system is based on Linux) and sends a command for running the tests (i.e. *adb shell am instrument -w -r -e* …). If all the test files are selected, the device iterates on each file and, inside each file, on each test method.
- For the unit tests, the IDE directly calls the *gradlew test* command on the folder where the project is located (there can be variations of the command). It launches a compilation of the project and the execution of the tests under the *test* folder. These test cases do not need any connected device.

© 2021 John Wiley & Sons, Ltd

DOI: 10.1002/stv

Figure 2. Organization of tests in *WordPress*.

There are frameworks, such as *Robolectric* [8], that allow to execute instrumented test cases without any connected device (physical or emulated). *Robolectric* also mentions the execution time as its main reason for existence: 'Running tests on an Android emulator or device is slow! Building, deploying, and launching the app often takes a minute or more'. The main problem with this framework is that it does not support all the functionalities of real devices.

3. BACKGROUND

This section reviews some relevant works related to mutation testing on mobile software, as well as the main cost-reduction techniques in mutation testing.

The massive development of software for mobile devices is so recent, and the platforms and operating systems evolve so quickly, that the techniques and tools that could be valid a few years ago may not be longer applicable today: as Kirubakaran and Kasthikeyani pointed out, 'By the time this paper has been presented, the mobile app landscape will have changed' [9].

3.1. Mutation testing on mobile apps

The special characteristics of mobile software (Table I) have a direct influence on testing.

Since mutation operators inject common errors that a competent programmer could introduce [12], the most significant research works about mutation testing of mobile software focus on: (1) analysing the suitability of classic mutation operators to mobile software and (2) the proposal of new mutation operators to reproduce common faults in this environment. Thus, Deng *et al*. [4] and Escobar-Velásquez *et al*. [5] follow this approach. Jabbarvand and Malek [13] directly propose mutation operators for the testing of the behaviour of apps with different energy-consumption rates.

Deng *et al*. [4] propose the 11 mutation operators for Android shown in Table II, that they classify into five categories. Their source of faults is some Google's technical documentation for testers and some significant characteristics of Android apps (event-driven nature, configuration saved in XML files, null values and screen orientation).

© 2021 John Wiley & Sons, Ltd

DOI: 10.1002/stv

Table I. Special characteristics of mobile software (taken from other studies[5, 10–11]).

(1) Connectivity and mobility in multiple network connections with different bandwidths.
(2) Different screens sizes, resolutions and orientations.
(3) Resource constraints (memory and processor).
(4) Context awareness and multiple input channels (users, sensors and networks).
(5) Potential interaction with other applications.
(6) Security and vulnerability.
(7) Finite energy source.
(8) Double nature of apps (native and web).
(9) Short development life cycle (to gain competitive advantage).
(10) Performance.
(11) Multiple devices and operating systems.

Table II. Android mutation operators [4].

Deng *et al.*'s mutation operators	
Category	Operator
Intents	Intent Payload Replacement
	Intent Target Replacement
Activity lifecycle	Lifecycle Method Deletion
Event handler	OnClick Event Replacement
	OnTouch Event Replacement
XML	Activity Permission Deletion
	Button Widget Deletion
	EditText Widget Deletion
	Button Widget Switch
Common faults	Fail on Null
	Orientation Lock

In a very recent article, Escobar-Velásquez *et al*. [5] extended a previous study of 2017 [14]. They analysed 2,023 fault reports taken from six different sources (bug reports of open source Android apps, bug-fixing commits of Android open source apps, Android-related Stack Overflow discussions, the Exception hierarchy of the Android APIs, Crashes and bugs described in previous studies and Reviews poster by users of Android apps on the Google Play Store).

Their analysis shows that 65% of the bugs are typical of any Java application, and that the remaining 35% are directly related to Android-specific characteristics. They classify the bugs using a taxonomy with 14 high-level categories of faults. Some categories (e.g. 'Collections and Strings') only contain Java faults, others (e.g. 'Activities and Intents') Android-specific faults and others (such as 'Input/Output') contain a mix of both.

The authors propose 38 mutation operators covering 10 out of the 14 categories. Some of the operators are specifically designed for Android.

Besides the taxonomy and the operators, an additional and interesting contribution is their *MutAPK* tool that directly inserts the faults into the compiled and packaged APK file. They also describe *MDroid+*, another tool that generates the faults from the source code.

Deng *et al*. [4] analyse the quality of their operators generating mutants for several apps and execute tests against the mutants. Escobar-Velásquez *et al*. [5] do not use any test case execution tool.

It is worth noting that Deng *et al*. emphasize the excessive cost of tests execution, because their mutant generation tool builds an *.apk* file for each mutant.

Jabbarvand and Malek [13] search 'energy anti-patterns', and from them, they build 50 operators that, for example, increase the frequency of the location update requests or do not switch off the Bluetooth. Their testing framework is called µDroid. The *µDroid* generates mutants, and to determine whether a mutant is killed, it compares its power consumption with the original program. There are no details about how test cases are executed. Regarding the test execution time, the authors only report about the mean times for determining whether mutants are killed (i.e. comparing

© 2021 John Wiley & Sons, Ltd

DOI: 10.1002/stv

the traces of energy consumption). The mean time is 11.7 s in the nine apps used in their experiments.

More recently, Paiva *et al*. [15] describe three mutation operators to test the specific behaviour of mobile applications related to the non-preservation of the UI state when apps are sent to background and then back to foreground. The iMPAcT tool is designed to deal with the mutants generated by these operators [16].

As it is seen, there is a common agreement related to the need of having specific mutation operators that reproduce common faults in Android applications.

However, to our best knowledge, and even though it is mentioned by several authors, there is a lack of work done in analysing, adapting or proposing the use of specific techniques to reduce the cost of mutation testing in this concrete environment. Deng *et al*. [4] is the only reference pointing out that performance should be improved with parallel execution, using fewer mutants or building a faster test framework. We analyse some of these options in this work.

3.2. Cost reduction techniques

Mutation testing has three main steps: mutant generation, test execution and result analysis. The most influencing factor in the global cost is the number of mutants, especially in the test execution step: the effort spent in mutant generation is almost insignificant with respect to test execution. Regarding result analysis, most tools give details about the mutation score, killed mutants per operator and so on.

So most cost reduction techniques focus on diminishing the number of mutants generated and on accelerating the test execution time. In this context, some of the most relevant techniques are as follows:

1. *Higher Order Mutation* is a form of mutation testing introduced by Jia and Harman [17]. This technique combines two or more mutants into the same mutated program (a higher order mutant). The empirical results by Polo *et al*. [18, 19] suggest that applying second-order mutants reduces the test effort by approximately 50%, without too much loss of test effectiveness. Langdon, Harman and Jia [20] built higher order mutants that are harder to kill than any first-order mutant. More recently, Abuljadayel and Wedyan [21] presented an approach to generate higher order mutants using a genetic algorithm, also harder to kill than first order mutants.
2. *Mutant Sampling* is a simple approach that randomly chooses a small subset of mutants from the entire set, according to a predefined percentage. Wong and Mathur [22] conducted an experiment using a variable selection rate x from 10% to 40%. The results of this study showed that Mutant Sampling is valid with an x value higher than 10%. Recently, Derezinska and Rudnik [23] proposed different mutant sampling criteria based on equivalence partitioning relative to object-oriented programme features. Based on the results, class random sampling and operator random sampling are recommended for OO in standard mutation testing, since the mutant sampling technique is easily applicable in comparison to other cost reduction techniques.
3. *Selective Mutation* [24, 25] firstly suggested by Mathur [26] and later extended by Offutt, Lee, Rothermel, Untch and Zapf [24] states that the number of mutants can be reduced by applying a subset of the mutation operators. If all mutants generated by the A mutation operator are also killed when the mutants produced by B are killed (i.e. one operator subsumes another one), then the tester could apply only one of these mutation operators, either A or B. The objective is to find a small set of mutation operators that generates a subset of all possible mutants without a major loss of test efficiency. Some of the most recent works in this line of research are previous studies [27–30].
4. *Parallel Execution* [31–32] distributes the mutants among different physical machines, executing test cases in parallel. It gets to reduce the total time of execution with no loss of effectiveness although it obviously requires a more complex infrastructure.
5. *Mutant Schema* [3] is designed to reduce the cost of test execution. The basic idea of this technique is to compose different programmes into a *metaprogram* (all the programme versions are

© 2021 John Wiley & Sons, Ltd

DOI: 10.1002/stv

included in a single file) that holds a set of *metaprocedures*. In order to determine which of the programme versions included in the schema must be executed, some type of control mechanism must be implemented. To our best knowledge, the first work about Mutant Schema is that of Untch, Offutt and Harrold *schema* [3], who created a *mutant schema generator* for Fortran. They used *metamutants* and *metaprocedures*. A metamutant contains all the mutants in a single file as a set of metaprocedures, which are functions that gather the different changes introduced by mutation operators.

That work from 1993 has inspired other researchers:

- Ma, Offutt and Kwon [33] adapt the idea to Java programmes in the MuJava tool, also automating the metamutant generation. These authors create metaprocedures for the object-oriented characteristics, such as inheritance, polymorphism and instantiation overhead. Some of these authors reuse this very same approach in a later work [34].
- Mateo and Usaola [35, 36] also apply Mutant Schema to Java programmes. They generate several metamutants (depending on the operators and the mutated class files) that include *if-else* statements to determine the version to be executed. The original Java bytecode is modified to call the corresponding mutant's metaprocedure.
- Papadakis and Malevris [37] apply the original Untch et al.'s approach, but adapting it to symbolic execution.

An additional cost-reduction technique is mutant generation at bytecode level. It consists in inserting the artificial faults directly on the bytecode, so avoiding the further compilation step. Some tools MuJava [33], Javalanche [38] and Bacterio [39]. A recent study by Hariri *et al*. [40] has shown that mutation testing at source level produces much fewer mutants than at bytecode level, so being test execution less expensive. Moreover, source level still generates a similar number of minimal and surface mutants, and the mutation scores at both levels are very closely correlated.

The techniques explored in this article are *Mutant Schema* and *Parallel Execution,* both isolated and in different combinations, as well as the non-use of any cost reduction technique at all. *BacterioWeb* supports *Mutant Sampling* too. *High Order* Mutation is not supported by the tool and, regarding *Selective Mutation*, *BacterioWeb* allows the tester to select the mutation operators to apply, but we have not made any analysis to check possible subsumptions among operators.

The strategy used for test execution also has a strong impact on the total testing time [6], [7]. In fact, in the most primitive model, the tester executes all test cases against all mutants, although it is possible to reduce the number of executions if each test case is only launched against those mutants remaining alive: suppose a system with seven mutants and five test cases. Suppose also that the killing matrix obtained after executing all tests against all mutants is the one appearing in Table III: as shown, 40 executions (five test cases against seven mutants plus the original programme) are required to complete the process.

However, if test cases are launched against mutants that, after each iteration, remain alive (i.e. the test suite does not attempt to 'kill twice' the same mutant), the number of executions may be lower while the mutation score is preserved: in Table IV, t_2 is not executed against the mutants that t_1 has already killed and, in general, t_{n+1} is not launched against the mutants killed by $t_1..t_n$. In this example, 20 executions are required (15 + 5 of the original).

Although launching 'all against all' requires in general much more executions, it may be useful if the tester desires to get a reduced test suite from the original one [41]: as shown in Table V, a greedy

Table III. An 'all against all' killing matrix for a supposed system.

Variable	m1	m2	m3	m4	m5	m6	m7
t_1	X			X	X		X
t_2	X			X	X	X	X
t_3	X			X	X		
t_4		X				X	
t_5		X	X			X	X

© 2021 John Wiley & Sons, Ltd

DOI: 10.1002/stv

Table IV. An ‘only against alive’ killing matrix for a supposed system.

Variable	m1	m2	m3	m4	m5	m6	m7
t_1	X			X	X		X
t_2						X	
t_3							
t_4		X					
t_5			X				

Table V. A reduced test suite obtained from Table III.

Variable	m1	m2	m3	m4	m5	m6	m7
t_1	X			X	X		X
t_5		X	X			X	X

algorithm applied to Table III would select $\{t_1, t_5\}$ as candidate test cases for the future regression tests.

After completing a cycle of mutation testing, *BacterioWeb* provides the tester a list of the best test cases applying a greedy algorithm such as the one just described.

4. MATHEMATICAL MODELS OF COST REDUCTION TECHNIQUES

This section models the theoretical savings of the cost reductions techniques used in the experimental section. These models are a basis for the experimentation and the evaluation of the combination of the cost-reduction techniques applied.

From the execution time point of view, the worst situation is when no cost-reduction technique is applied: neither Mutant Schema nor Parallel Execution. In this situation, all test cases are executed against all mutants. Although the absence of cost-reduction techniques is obviously unadvised, it is useful to take it as a baseline for estimating the cost reduction achieved. This idea is similar to that of Grindal, Offutt and Andler [42] in their paper about combinatorial test generation: although *All combinations* is not a good technique (it produces many test cases, many of which are redundant), ‘it is often used as a benchmark with respect to the number of test cases’.

Setting aside the result analysis step, the total time required for executing T (a set of test cases) against M (a set of mutants) is the sum of the times for mutant generation and for the required steps for executing the tests (Equation 1).

$$T = T_{gen} + T_{exec} \tag{1}$$

Both with or without Mutant Schema, and independently of the execution algorithm, the mutant generation time (T_{gen}) is equal and does not depend on the approach. So we will not consider it in the next equations.

4.1. Mathematical model of T_{exec} without any cost-reduction technique

With respect to the execution time, it depends on the following:

1 The number of test cases and the number of mutants ($|M|$).
2 The nature of the test suite (*unit* or *instrumented*). The execution of *instrumented* test suites requires compiling the app, packaging it into an *apk*, pushing it onto the device and executing the tests. *Unit* tests only need the compilation and the execution.
3 The execution algorithm: in this paper, we distinguish between executing all test cases *against all the mutants* (‘All against all’, such as in the example of Table III) and *only against the mutants remaining alive* (‘Only Alive’, like in Table IV).

© 2021 John Wiley & Sons, Ltd

DOI: 10.1002/stv

Table VI. Execution tasks depending on the type of tests.

Type of test	Tasks
Unit	Compile Run tests
Instrumented	Compile (and build APK) Push APK onto devices Install APK Run tests

Table VI summarizes the tasks to be performed depending on the type of test.

In the case of *instrumented* tests, it must build, push and install an *apk* with the change that corresponds to every mutant. The tests are launched against all the mutants (*M*), as shown in Equation 2. For *unit* test cases, $T_{push} = T_{install} = 0$.

$$T_{exec}^{NoTech} = |M| \cdot \left(T_{compile} + T_{push} + T_{install} + T_{run}\right) \tag{2}$$

In Equation 2:

- $T_{compile}$ is the time required to compile one version of an app.
- T_{push} is the time required for pushing the *apk* file corresponding to an app from the computer to the device.
- $T_{install}$ is the time required for installing an app on a device. The *apk* file has been previously deployed onto that device.
- T_{run} is the time required for executing the test suite against an app.

4.2. Mathematical model of T_{exec} with Mutant Schema

The same steps are required for Mutant Schema. There is however a previous step to generate and mount the schema (T_{ms}), but there is only one compilation and, for *instrumented* tests, only one pushing and one installation. As in the previous case (we are considering the *All against all* execution), all the tests are launched against all the mutants, as shown in Equation 3.

$$T_{exec}^{MS} = T_{ms} + T_{compile} + T_{push} + T_{install} + |M| \cdot T_{run} \tag{3}$$

Actually, T_{ms} is very-very low in most cases (7–8 ms in almost all projects). Even in the case of WordPress (one of the selected applications for our experiment, which has 538 Java source files and 109,991 lines of code), the generation of the Mutant Schema is almost insignificant (Figure 3). Thus, we will remove T_{ms} from our equations.

4.3. Mathematical model of Parallel Execution

Without Mutant Schema and with n devices, the execution time is directly reduced in $1/n$ (Equation 4): every task is made $|M|$ times, but distributed in parallel on the n connected devices.

$$\mathrm{T}_{exec,\,n}^{NoTech} = \frac{|M| \cdot \left(T_{compile} + T_{push} + T_{install} + T_{run}\right)}{n} \tag{4}$$

Mutant Schema still requires only one compilation; the system must be uninstalled and installed on all the devices, but since these tasks are executed in parallel, it is like performing them only once. T_{run} is reduced in $1/n$ (Equation 5):

© 2021 John Wiley & Sons, Ltd

Times (in milliseconds)		
Untch schema mounting	APK build	
542	**51699**	**<— Total time**
54	***5170***	**<— Mean time**
58	5085	
55	5189	
53	5299	
51	5417	
53	5415	
52	4958	
52	5402	
50	4999	
67	4926	
51	5009	

Figure 3. Ten measures of T_{ms} and $T_{compile}$ for WordPress and its 538 Java files.

$$T^{MS}_{exec,\,n} = T_{compile} + T_{push} + T_{install} + \frac{|M| \cdot T_{run}}{n} \tag{5}$$

Equations 4 and 5 substitute Equations 2 and 3 with the introduction of the new parameter, n (n = 1 when there is no parallel execution).

4.4. *Mathematical model of Only Against Alive with Mutant Schema*

The success of Only Against Alive depends on the better or worse 'luck' in the execution order of the test cases:

- The best situation happens if the first test case is able to kill all the mutants, since no more test cases need to be executed (i.e. the killing matrix would have only one row with all the mutants killed). This is illustrated in Table VII, where the first test finds all the artificial faults. In this case, there is a cost reduction factor (we call it ρ) of $1/|T|$.
- The worst situation is when none of the tests kills any mutants (all the cells in the matrix would be empty) or if the last test case executed is the only one that kills mutants (Table VIII). In any

Table VII. The first test kills all the mutants (ρ = 1/5).

Variable	m1	m2	m3	m4	m5	m6	m7
t_1	X	X	X	X	X	X	X
t_2							
t_3							
t_4							
t_5							

Table VIII. The last test kills all the mutants (ρ = 5/5).

Variable	m1	m2	m3	m4	m5	m6	m7
t_1							
t_2							
t_3							
t_4							
t_5	X	X	X	X	X	X	X

© 2021 John Wiley & Sons, Ltd

DOI: 10.1002/stv

of these two situations, all the tests are executed against all the mutants. There is no cost reduction in this case, so $\rho = 1$.

Without Mutant Schema, the total time is given by Equation 6:

$$T_{exec,\,n}^{NoTech,\,OA} = \frac{|M| \cdot \left(T_{compile} + T_{push} + T_{install} + \rho\, T_{run}\right)}{n} \tag{6}$$

With Mutant Schema, the total time is (Equation 7):

$$T_{exec,\,n}^{MS,\,OA} = T_{compile} + T_{uninstall} + T_{install} + \frac{|M| \cdot \rho \cdot T_{run}}{n} \tag{7}$$

In the *worst case*, the reduction factor is $\rho = 1$, being in this case the times equal to those in Equations 4 and 5.

In the *best case*, the reduction factor is $\rho = 1/|T|$, where $|T|$ is the number of test cases. This behaviour occurs if all the mutants are killed by the first test case executed.

In the *average cases*, the reduction factor takes intermediate values. Therefore, $\frac{1}{|T|} \leq \rho \leq 1$.

Obviously, every test case requires a different time for running, and therefore, the reduction factor ρ is not exactly $1/|T|$. Consider however that we are executing the same test cases against hundreds or thousands of versions (the mutants) of the SUT, and in the experimentation, every test cycle has been executed several times. Thus, we can accept that the mean times are normally distributed and that the reduction factor is, in general, very proximal to $1/|T|$. Anyway, Table XIII will show these very small differences.

4.5. Improvement factor

The improvement in the execution time with respect to our benchmark (non-using any technique, Equation 2) can be described as a quotient. Thus, the 'improvement factor' (*IF*) of applying Mutant Schema with Parallel execution on n devices and an arbitrary reduction factor (ρ) is given by Equation 8. Note that, if $\rho = 1$, this equation is valid for the *All against all* execution algorithm.

$$IF = \frac{T_{exec,\,n}^{NoTech,\,OA}}{T_{exec,\,n}^{MS,\,OA}} \tag{8}$$

We develop the equation replacing 4 and 5 in 8:

$$IF = \frac{\dfrac{|M| \cdot \left(T_{compile} + T_{push} + T_{install} + \rho \cdot T_{run}\right)}{n}}{T_{compile} + T_{push} + T_{install} + \dfrac{|M| \cdot \rho \cdot T_{run}}{n}} \tag{9}$$

Removing n from the numerator:

$$IF = \frac{|M| \cdot \left(T_{compile} + T_{push} + T_{install} + \rho \cdot T_{run}\right)}{n \cdot \left(T_{compile} + T_{push} + T_{install}\right) + |M| \cdot \rho \cdot T_{run}} \tag{10}$$

© 2021 John Wiley & Sons, Ltd

DOI: 10.1002/stv

As $|M|$ grows up, *IF* improves, although it tends to asymptotically stabilize towards a maximum (Equation 11).

$$\lim_{|M|\to\infty} IF = \frac{T_{compile} + T_{push} + T_{install} + \rho \cdot T_{run}}{\rho \cdot T_{run}} = 1 + \frac{T_{compile} + T_{push} + T_{install}}{\rho \cdot T_{run}} \quad (11)$$

4.6. Assumptions

The comparison of the times and the calculus of the limit in Equation 11 requires that the different times can be compared. Thus, we assume that all of them are similar: that is, the times required to compile ($T_{compile}$), pushing (T_{push}) and installing ($T_{install}$) an app are similar with or not without Mutant Schema or Parallel Execution. In the same way, we assume that T_{run}, which is the time required to execute 1 test case against an app is also similar, independently on the use of Mutant Schema.

These assumptions are discussed in Section 6 (Research Question 1).

5. MUTANT SCHEMA GENERATION

The Mutant Schema are generated from the source code following the Untch *et al*.'s idea [3].

Every Java source file is processed with the *javaparser* library [43]. This library builds the abstract syntax tree of each processed file. Both the source code and the serialized abstract syntax tree are saved in a MongoDB database (the general architecture of *BacterioWeb* will be described in Section 6.2).

The Mutant Schema generator iterates trying to apply each selected mutation operator to the considered file. For example, the traditional AOR operator takes all the binary expressions in the file and, if the corresponding operator is +, −, *, /or %, modifies the original statement by a call to *MutantDriver. X*, where *X* is the name of original operator.

Consider the statement $a + b + c$: in prefix notation it can be written as $+(a, +(b, c))$. If we substitute the operator by a call to a *PLUS* method in a *MutantDriver*, the statement can be rewritten as follows:

```
MutantDriver.PLUS(a, MutantDriver.PLUS(b, c))
```

In this case, every operator can be replaced by the other four operators. This is: + is replaced by −, *, / and %; − is replaced by +, *, / and %, and so on.

Suppose the first mutated binary expression is $b + c$. In order to represent the four possible mutants, the statement *MD. PLUS(b, c)* will be written as follows:

```
MutantDriver.PLUS(b, c, 1, 2, 3, 4)
```

The four indexes (1–4) reference the *mutant index*.

Then, the second binary expression ($a + b + c$, that now is a + *MutantDriver.PLUS(b, c, 1, 2, 3, 4)*) is mutated. The whole expression remains as follows:

```
MutantDriver.PLUS(a, MutantDriver.PLUS(b, c, 1, 2, 3, 4), 5, 6, 7, 8)
```

The *MutantDriver* implements the *PLUS (int x, int y, int … indexes)* metaprocedure as the method shown in Figure 4: the first two parameters are the numbers to be added; the others are the indexes of the applicable mutants (note that Java allows to pass a variable number of parameters with the suspension points, which can be processed as an array).

Consider the following situations:

a We are executing the test suite against the original program, which has 0 as mutant index: in this case, the implementation of *PLUS* reads the value of *currentMutant* from a file. Since its value is zero, it returns the result pointed by the first *if:* $a + b$, which is the same than in the original programme.

© 2021 John Wiley & Sons, Ltd

DOI: 10.1002/stv

```
public static int PLUS(int a, int b, int... indexes) {
  loadCurrentMutant();
  int location =
        Arrays.binarySearch(indexes, currentMutant);

  if (currentMutant == 0 || location < 0) return a + b;
  if (location == 0) return a - b;
  if (location == 1) return a * b;
  if (location == 2) return a / b;
  if (location == 3) return a % b;
  return a + b;
}
```

Figure 4. Implementation of *PLUS* in the *MutantDriver*.

b We are now executing the mutant index with value 3 that is the third mutant proceeding from $b + c$. This value has been saved in the aforementioned file and is assigned to *currentMutant* in the *loadCurrentMutant* method. This value (3) is searched in the array passed in the variable parameters set (it was in previous studies [1, 2, 24, 44]) and found with $location = 2$. Then, the method returns *a/b*.

c If we are executing the test suite against the mutant index number *100*, since this value is not found in the array, the method returns the expression in the first *if*.

As an additional example, Figure 5 shows the implementation we have given to the *ITR* mutation operator described by Deng *et al*. [4] and by Escobar-Velásquez *et al*. [5].

The *ITR* operator replaces the target activity of an *Intent*. Consider the statement:

```
Intent i = new Intent(this, MainActivity.class)
```

Supposing the mutant to be generated is the 15th, the statement is changed to:

```
Intent i = MD.ITR(this, MainActivity.class, 15)
```

When the mutant is to be applied (i.e., *currentMutant==15),* the program will behave as if the programmer would have written:

```
Intent i = new Intent(this, Activity.class)
```

5.1. *Disadvantages of Mutant Schema (I): triviality of schema mutants*

Often, schema mutants are much easier to kill than traditional ones. Consider the small, selected, fragment of the *Kuar* app (one of the projects used in our experiments) shown in Figure 6: a *SlidingBoard* holds a collection of *Square* instances.

```
public static Intent ITR(Context ctx, Class<?> activityClass, int index) {

    loadCurrentMutant();
    if (index == currentMutant)
        return new Intent(ctx, Activity.class);
    return new Intent(ctx, activityClass);
}
```

Figure 5. Implementation of the *ITR* operator in the *MutantDriver*.

© 2021 John Wiley & Sons, Ltd

DOI: 10.1002/stv

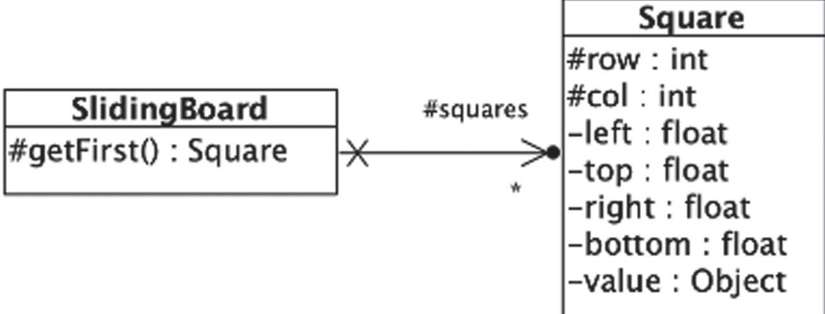

Figure 6. A small excerpt of the *Kuar* design.

```
if (squares[row][col].getValue() != null &&
  (Integer) squares[row][col].getValue() == 1) {
          ...
}
```

Figure 7. A fragment of the code in the *SlidingBoards' getFirst* method.

```
if (MutantDriver.AND(squares[row][col].getValue()!= null,
       (Integer) squares[row][col].getValue() == 1, 2)) {
          ...
}
```

Figure 8. One of the schema mutants of *getFirst* in *SlidingBoard*.

```
Original program

if (x >= board.getLeft() && x <= board.getRight() && y >= board.getTop() && y <=
board.getBottom()) {
```

```
LOR mutant

if (x >= board.getLeft() || x <= board.getRight() && y >= board.getTop() && y <=
board.getBottom()) {
```

```
ROR mutant

if (x >= board.getLeft() && x != board.getRight() && y >= board.getTop() && y <=
board.getBottom()) {
```

```
Mutant Schema combining LOR and ROR

if
MutantDriver.AND(MutantDriver.AND(MutantDriver.AND(MutantDriver.GREATER_EQUALS(x,
board.getLeft(), 214, 215, 216, 217, 218), MutantDriver.LESS_EQUALS(x,
board.getRight(), 219, 220, 221, 222, 223), 176), MutantDriver.GREATER_EQUALS(y,
board.getTop(), 224, 225, 226, 227, 228), 177), MutantDriver.LESS_EQUALS(y,
board.getBottom(), 229, 230, 231, 232, 233), 178)) {
```

Figure 9. A piece of code, two classic mutants and a Mutant Schema.

Figure 7 shows a small piece of code of the *getFirst* method of the *SlidingBoard* class. Note that it takes the *Square* instance located at the coordinates *(row, col)* and, if it is not *null*, reads its value and casts it as an *Integer*. Since the whole decision is evaluated in short-circuit, if the instance is *null*, the second condition is not evaluated.

© 2021 John Wiley & Sons, Ltd

DOI: 10.1002/stv

The statement in Figure 7 works rightly both in the original programme as in a classic mutant that replaces, for example, the *&&* by applying the *LOR* operator.

Consider however the effect produced by the same operator with Mutant Schema, that appears in Figure 8: the original infix expression (with the form *A && B)* is translated into a prefix expression *MutantDriver.AND(A, B, 2),* where the last value references the mutant index.

In this second case, the decision is not evaluated in short-circuit and, then, both conditions are evaluated. Suppose that the *Square* instance is *null:* in the first time, the instance is compared to *null*. With independence of the result returned, the second condition is checked and, since the instance is *null*, the program cannot cast it to an *Integer* and crashes, producing what is usually known as a *trivial mutant* [5].

According to Escobar-Velásquez et al. [5], a *trivial mutant* is a mutant that always or frequently crashes at runtime. Actually, trivial mutants introduce 'noise' in the result analysis phase and may lead to misinterpret the mutation score.

In general, these operators are more frequent with schema than with traditional mutants, although Deng *et al*., who do not use Mutant Schema, also are aware of the problem they represent ('we need to make our tool generate fewer mutants that immediately crash [...]').

5.2. Disadvantages of Mutant Schema (II): legibility of schema mutants

The example in Figure 9 proceeds from the *Kuar* app too: the original statement checks whether a point is inside the area of a board. The statement is a decision with four conditions and is mutable in several ways.

Suppose that the tester wants to investigate why one of this statement's mutants remains alive: clearly, this task is much easier comparing the original code in rows 2 and 3, than with the too confusing call to the Mutant Driver of the last row.

6. EXPERIMENTATION

In this section, we combine three cost reduction techniques (*Mutant Schema*, *Parallel Execution* and *Only Alive*) and analyse their influence on the execution time of mutation testing in mobile software. The benchmark for doing the comparison is the non-use of any cost-reduction technique. From these experiments, we check the goodness of our mathematical models to estimate the execution cost of a mutation testing cycle.

It is important to remind that the goal of this paper is neither the proposal nor the validation of mutation operators, but only measuring how the application of different techniques reduces the time spent in mutation testing. Anyhow we use both traditional mutation operators as Android-specific.

6.1. Research questions

In these experiments, we assess the impact of using several cost reduction techniques in mutation testing with a focus on mobile applications. We also validate the mathematical models presented in Section 4.

Our research questions are as follows:

- RQ1: Can we accept the assumptions made in the Epigraph 4.6 (assumptions) of Section 4 (mathematical models) valid? This is, the times for compiling, pushing, installing and running are the same, independently of the use or not of Mutant Schema?
- RQ2: How good are the mathematical models to predict the mutation testing time?
- RQ3: How does the number of mutants influence on the improvement factor?

6.2. Mutation testing tool

The tool used for performing the experiments is *BacterioWeb*, a mutation tool we have developed as an evolution of *Bacterio* [39]. *BacterioWeb* runs on a web server and can be executed with any browser. The user (a tester) uploads his/her Android projects to the server. Figure 10 describes, as use cases, the main functionalities of the tool and its relationships with external actors: the

© 2021 John Wiley & Sons, Ltd

DOI: 10.1002/stv

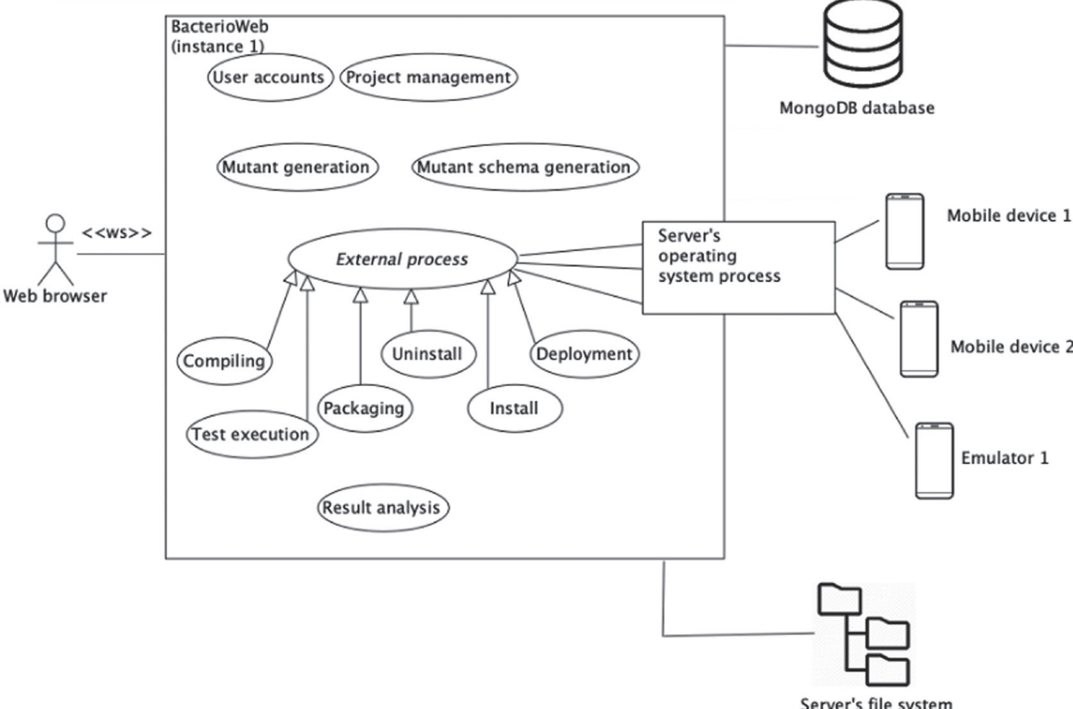

Figure 10. General structure of *BacterioWeb*.

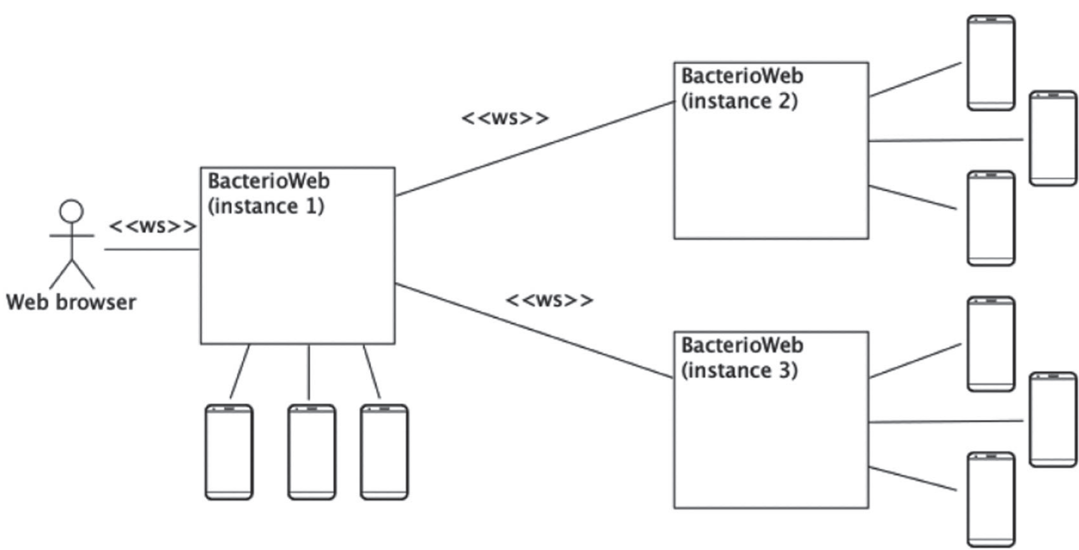

Figure 11. Collaboration among different instances of *BacterioWeb*.

projects are saved in MongoDB databases (therefore making the projects accessible from any place) and in the server's local file system. The communication with the external devices (physical devices or emulators) is achieved by means of the creation of operating system processes. For example, the execution of a command on a device requires creating an operating system process for sending the order. All communication between the frontend and the backend is based on the *websocket* protocol to keep the user informed of the progress of mutant generation, test execution and so on.

© 2021 John Wiley & Sons, Ltd

DOI: 10.1002/stv

Moreover, and in order to push forward the parallelism, *BacterioWeb* may communicate with other instances of *BacterioWeb* running on other servers, and this in turn helps the use of the devices they are not using at some point: in Figure 11, instance 1 can send mutants and test cases for execution on the devices connected to instances 2 and 3.

6.3. Target Android apps

The experiments have been run on the following eight Android apps, which were selected according to the following criteria:

- To make use of common mobile functions (such as touch events).
- They must have available test cases implemented by the developers.
- Their source code must be also available.

1. *WordPress* is the biggest application used in this study. It is used for creating web sites and blogs. It is published in the Google Play Store, it has over 10 million downloads and almost 150,000 comments. Its source code is available at github.
2. *Figures* is a project implemented during the development of *BacterioWeb* for testing some of its characteristics. It is an app that calculates the perimeter and type of a triangle or of a quadrilateral. The calculus can be done locally or by querying a remote web service (Figure 12a). Moreover, the lengths of the sides can be: (1) directly written (Figure 12b), (2) calculated from the coordinates where the tester clicks (Figure 12c) or (3) calculated by clicking on the measures from different sensors (Figure 12d). The selected combination is saved in a set of preferences. It is not a complex project, but it holds the logic of the triangle-type determination

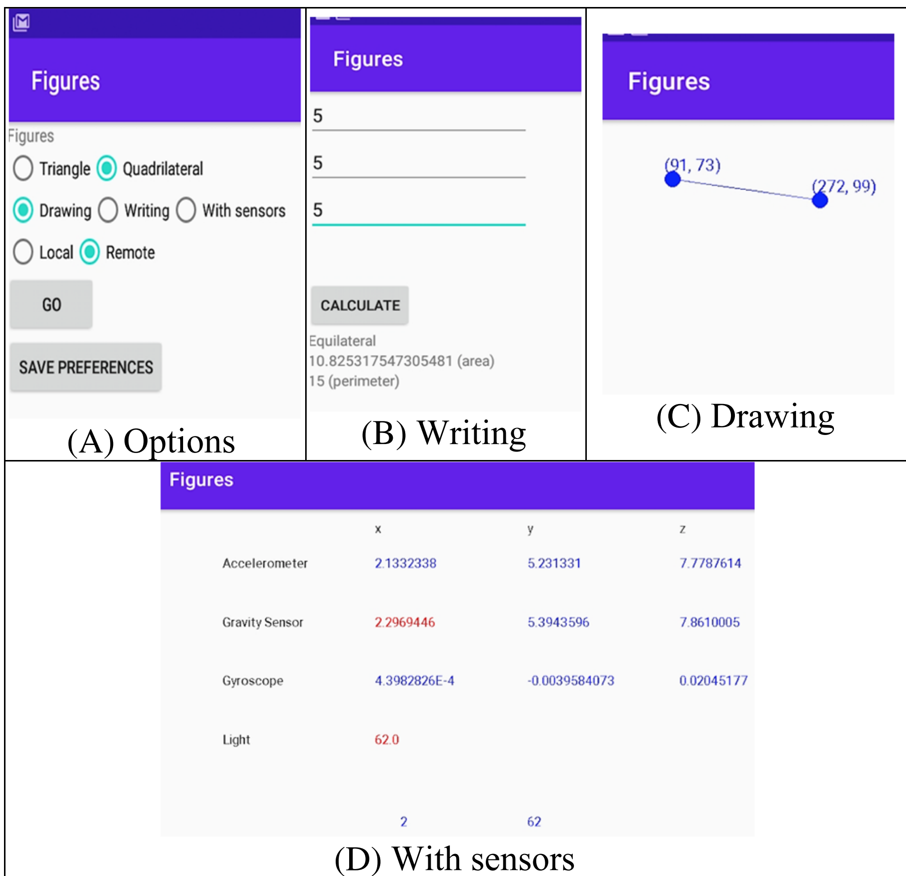

Figure 12. Screenshots of *Figures*.

© 2021 John Wiley & Sons, Ltd

DOI: 10.1002/stv

Table IX. Some characteristics of the apps.

App	Mutated classes		#	
	#Files	LOC	Tests	Mutants
WordPress	538	109,991	120	29,554
Figures	13	6,450	31	1,331
Alarm Klock	92	6,608	12	3,239
JustSit	4	483	10	171
Tourism	54	4,902	15	851
Mangosta	59	9,902	31	1,579
Dexter	30	2,872	10	216
Kuar	37	3,580	12	3,225
Total	**827**	**144,788**	**241**	**40,166**

problem (proposed by Myers [45] and typically used in many papers about testing), dealing with user events, with web communication, with sensors' data and with preferences.

3 *AlarmKlock* and *JustSit* are two of the apps used in Deng et al.'s [4] work on mutation operators for Android. *AlarmKlock* allows to set up alarms for different days and hours.
4 *JustSit* is a single app that shows the user two time counters, one in seconds and the other in minutes.
5 *Tourism* has been developed by a company for a Spanish regional government. It uses Google services to plan touristic visits to cities. The user adds points of interest to a touristic route and constraints the time and budget; the app gives in return a route fitting the user's input.
6 *Mangosta* is an open source, open standard, XMPP/Jabber client. Its code is available at github.
7 *Dexter* is an Android library that simplifies the process of requesting permissions at runtime. It is also available at github.
8 *Kuar* is a game developed some years ago by one of the authors of this article. The user must consecutively order the numbers on a board in the fewest moves.

Table IX shows some quantitative data of the apps: the *#Files* and *LOC* columns correspond to the number of Java files in each project (excluding interfaces) and their number of lines of code. Last columns contain the number of test cases in each project and the maximum number of mutants that *BacterioWeb* can generate. We wrote the test cases for *Figures*, *Kuar* and *JustSit*. Test cases for *Tourism* were provided by its developer. The other projects have test cases in their respective repositories.

6.4. Server hardware and mobile devices

For this experiment, only one instance of *BacterioWeb* is used. Both the Tomcat server and the MongoDB instances are served from localhost by a MacBook Pro with 16 Gb RAM. Anyway, the times of connection to the database, writing, reading and so on are removed from the time calculus.

There are two identical physical devices connected to the server: two *Samsung* tablets, model SM-T590 running Android 8.1.0 with 3 Gb RAM.

6.5. Applied mutation operators

As we have said, our goal is not to validate mutation operators for Android, but to measure the influence of several techniques in the reduction of the time required for mutation testing in Android projects. Obviously, using one or another operator with so many mutants and executions does not have any significant impact on the results of our experiments, which are not concerned with the quality of operators, but with the time required for mutation testing. However, we have implemented the traditional operators most used in the literature for java projects [33] and some Android-specific operators proposed by Deng *et al*. [4] and Escobar-Velásquez *et al*. [5]. The set

© 2021 John Wiley & Sons, Ltd

of operators will introduce both traditional errors (typical of the Java language) as specific errors for Android (as event handlers).

Traditional operators:

LOR (Logical Operator Replacement), which substitutes a logical operator by another one (|| by &&, | and so on).

ROR (Relational Operator Replacement), which replaces a relation operator by another one (e.g. && by ||).

UOI (Unary Operator Insertion), which inserts predecrements, postdecrements and the unary minus in numeric variables.

AOR (Arithmetic Operator Replacement), that replaces some arithmetic operators by others (+ by –, *, /, and so on).

SVR (Scalar Value Replacement) replaces a variable value by a constant. Our implementation replaces strings by the empty string.

IMCA (Invalid Method Call Argument) is one of the Escobar-Velásquez *et al*.'s operators [5]. It randomly mutates a method call argument of a basic type.

Android mutation operators:

The Android operating system makes available to developers different mechanisms for storing data in files: *SharedPreferences* files are key-value tables that allow the application to store small data sets in a simple way. All changes made in an editor are batched, and not copied back to the original *SharedPreferences* until a call to *commit()* or *apply()* is executed. We define three mutation operators to reproduce errors that can occur when working with *SharedPreferences* files:

FEC (Forget Editor Commit): This operator simulates that the developer forgets calling *commit*. The values are set with some of the *putX* methods, but the changes are not materialized. So this operator removes the statement *editor.commit()*. For killing these mutants, test cases need either to include an oracle to check that the preferences have been save (what is unusual), or to execute a long scenario that makes use of the previously saved preferences: therefore, this operator confirms one of the conclusions of Fraser and Gargantini in [46], who observed that is preferred to have a few long test cases than many short test cases. This operator is quite similar to the *CPSE* operator proposed by Paiva *et al*. [15].

FEA (Forget Editor Apply): This operator is similar to FEC, but in this case, it removes the call to the *apply()* method. It works exactly in the same way as the FEC mutation operator.

RSPE (Replace Shared Preferences Editor): A typical error in the use of *SharedPreferences.Editor* type files is to mismanage the keys entered in the file. This operator mutates the key-value pairs of the statements *putInt(…)*, *putBoolean(…)*, *putString(…)*, *putLong(…)*, *putFloat(…)* of different ways.

MDL (Lifecycle Method Deletion) is one of Deng *et al*.'s operators [4]. It deletes each overriding activity method to force Android to call the version in the super class.

ETR (OnTouch Event Replacement) [4]: It searches and stores all event handlers that respond to *OnTouch* events in the current class. Then, it replaces each handler with every other compatible handler.

KER (Key Event Replacement): Key events contain information about keys pressed. This operator replaces some key events with other equivalent key events: for example, it replaces *KeyEvent.ACTION_UP* by *KeyEvent.ACTION_DOWN*.

IEC (Interchanges the Event's Coordinates): This operator modifies motion event's location through interchanges and replacement of axis values. So, if the user clicks on (100, 200), this mutation operator sends the event to (200, 100).

IPR_E (Intent Payload Replacement Extension): This operator is an extension of the IPR operator (Intent Payload Replacement) proposed by Deng *et al*. [4]. An *Intent* can send different types of data from one activity to another, as key-value pairs. The *putExtra(…)* method takes the key name as the first parameter, and the value as the second parameter. IPR_E includes all mutations of IPR proposed in [4], but it also adds a mutation for the first parameter (*empty String*) and different mutations for the key-string pairs.

ITR (Intent Target Replacement) [4]**, also called InvalidActivityName** in Escobar-Velásquez *et al* [5]: This operator mutates the *Intent* target object (an activity), changing the target activity. This idea also is included in the *NACT* operator proposed by Paiva [15].

© 2021 John Wiley & Sons, Ltd

ORL_M (Orientation Locked Modified): This operator is a modification of the Deng *et al*.'s ORL operator [4]. The original ORL freezes the orientation of an activity to be in portrait or landscape, and this is done by inserting a locking statement into the source. Our modification preserves the same idea, but the mutants freeze the orientation of an activity to be in portrait or landscape through insertion or replacement of *setRequestedOrientation(…)* statement into the source.

MJP (Modify JSON Put): This operator inserts small changes into the key-value pairs of the different *put(...)* methods of the *JSONObject* class. JSON is a widely used format for message interchange. Developers tend to copy and paste calls to *put(...)* or to build unexpected hierarchies of JSON objects. This operator modifies keys and values in *put* calls.

DICO (Incorrect Call of Opening): SQLite databases can be opened either in *read* or in *read/write* mode. This operator replaces calls to *getWritableDatabase* by *getReadableDatabase* in *SQLiteOpenHelper* objects.

DIOM (Incorrect Opening Method): This operator has the same effect that DICO, but replacing the flag corresponding to the database opening mode on *SQLLiteDatabase* objects: it changes *OPEN_READWRITE* by *OPEN_READONLY* in the *openDatabase* method.

IQ (Incorrect Query): The SQLite database allows the developer to introduce SQL statements as strings. This operator mutates the query passed as parameter in calls to *SQLiteDatabase.RawQuery(…)*. This operator is similar to the *InvalidSQLQuery* proposed by Escobar-Velásquez *et al*. [5].

RAQ (Replace read-write Access to a database Query): This operator mutates the calls allowing the iteration through the result set returned by a database query. It changes *moveToFirst*, *moveToLast*, *moveToNext* and *moveToPrevious* by the others.

SIR (Service Identifier not Returned): This operator mutates the *getSystemService (java.lang.String)* method, replacing the name of the service required by *null*.

6.6. Experimental method

For each application under test, we have carried out the following steps:

- *Mutant generation*. This task generates and saves the mutants in the MongoDB database.
- Second, we execute the test suites against the mutants with the combinations of techniques shown in Figure 13. Excepting for WordPress, each test suite has been executed against all the mutants three times to minimize bias. For WordPress, we have taken a sample (10%) of mutants, since otherwise the execution time is huge. Maybe three times does not seem too much, but some complete executions with the combination 1 require around a week. At this point, it is important to note that, since overheating has a very negative impact on computer performance, all the executions have been made in a room with fixed temperature: the server (Epigraph 6.2) is a simple personal computer that has been subjected to a very heavy workload for many hours.

BacterioWeb saves many information in a set of comma-separated files: in one of them (*global.txt)*, it accumulates all the execution data of all the projects; in the other, it saves exactly the same data, but creating a single file for each test suite execution. Figure 14 shows an excerpt of *global.txt*: each row holds the data of the execution of one test case against one mutant. The columns contain: (A) the unique *id* of this test suite execution, (B) the project under test, (C) the test

1	(NoTech, All against all, 1 device)
2	(NoTech, Only alive, 1 device)
3	(NoTech, All against all, 2 devices)
4	(NoTech, Only alive, 2 devices)
5	(Mutant Schema, All against all, 1 device)
6	(Mutant Schema, Only alive, 1 device)
7	(Mutant Schema, All against all, 2 devices)
8	(Mutant Schema, Only alive, 2 devices)

Figure 13. Execution combinations.

© 2021 John Wiley & Sons, Ltd

DOI: 10.1002/stv

executor used (*NoMS* or *MS*), (D) the test case, (E) the mutant index, (F) the current class under test, (G) the verdict (*A* or *K* depending on whether the mutant is alive or killed), (H) the test case type, (I) the device where the mutant has been installed, the date (J) and time (K), the execution algorithm (*All against all* or *Only against alive)* in column *L*, the compile, push and install execution times (M, N and O) and the run time (P) required by this test case with this mutant.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1	ExecutionId	Project	Mutant	TestCase	MutantIn	Class under test	Verdic	TestType	Device	Date	Time	Algorithm	BuildAPKT	PushTime	InstallTim	Trun
300	30f39946-4	mangosta-a	NoMS	...CreateChatActivityInstrumen	117	inaka.com.mangosta.activitie:	K	androidTe	4bd3f236	20/5/20	9:23:03 a. m.	againstAlive	0	0	0	90
301	30f39946-4	mangosta-a	NoMS	...CreateChatActivityInstrumen	129	inaka.com.mangosta.activitie:	A	androidTe	94769a87	20/5/20	9:23:10 a. m.	againstAlive	0	0	0	102
302	30f39946-4	mangosta-a	NoMS	...EditChatMembersActivityIns	131	inaka.com.mangosta.activitie:	A	androidTe	4bd3f236	20/5/20	9:23:23 a. m.	againstAlive	3812	926	5063	94
303	30f39946-4	mangosta-a	NoMS	...CreateChatActivityInstrumen	129	inaka.com.mangosta.activitie:	A	androidTe	94769a87	20/5/20	9:23:24 a. m.	againstAlive	0	0	0	138
304	30f39946-4	mangosta-a	NoMS	...CreateChatActivityInstrumen	129	inaka.com.mangosta.activitie:	K	androidTe	94769a87	20/5/20	9:23:33 a. m.	againstAlive	0	0	0	90
305	30f39946-4	mangosta-a	NoMS	...EditChatMembersActivityIns	131	inaka.com.mangosta.activitie:	A	androidTe	4bd3f236	20/5/20	9:23:34 a. m.	againstAlive	0	0	0	113
306	30f39946-4	mangosta-a	NoMS	...EditChatMembersActivityIns	131	inaka.com.mangosta.activitie:	A	androidTe	4bd3f236	20/5/20	9:23:44 a. m.	againstAlive	0	0	0	105
307	30f39946-4	mangosta-a	NoMS	...EditChatMembersActivityIns	138	inaka.com.mangosta.activitie:	A	androidTe	94769a87	20/5/20	9:23:54 a. m.	againstAlive	4250	876	5315	96
308	30f39946-4	mangosta-a	NoMS	...EditChatMembersActivityIns	131	inaka.com.mangosta.activitie:	A	androidTe	4bd3f236	20/5/20	9:23:56 a. m.	againstAlive	0	0	0	112
309	30f39946-4	mangosta-a	NoMS	...EditChatMembersActivityIns	131	inaka.com.mangosta.activitie:	A	androidTe	4bd3f236	20/5/20	9:24:05 a. m.	againstAlive	0	0	0	95
310	30f39946-4	mangosta-a	NoMS	...EditChatMembersActivityIns	138	inaka.com.mangosta.activitie:	A	androidTe	94769a87	20/5/20	9:24:05 a. m.	againstAlive	0	0	0	117
311	30f39946-4	mangosta-a	NoMS	...EditChatMembersActivityIns	138	inaka.com.mangosta.activitie:	A	androidTe	94769a87	20/5/20	9:24:16 a. m.	againstAlive	0	0	0	105
312	30f39946-4	mangosta-a	NoMS	...BlockUsersActivityInstrumen	131	inaka.com.mangosta.activitie:	A	androidTe	4bd3f236	20/5/20	9:24:16 a. m.	againstAlive	0	0	0	111
313	30f39946-4	mangosta-a	NoMS	...BlockUsersActivityInstrumen	131	inaka.com.mangosta.activitie:	A	androidTe	4bd3f236	20/5/20	9:24:27 a. m.	againstAlive	0	0	0	108
314	30f39946-4	mangosta-a	NoMS	...EditChatMembersActivityIns	138	inaka.com.mangosta.activitie:	A	androidTe	94769a87	20/5/20	9:24:27 a. m.	againstAlive	0	0	0	114
315	30f39946-4	mangosta-a	NoMS	...EditChatMembersActivityIns	138	inaka.com.mangosta.activitie:	A	androidTe	94769a87	20/5/20	9:24:37 a. m.	againstAlive	0	0	0	97

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1	ExecutionId	Project	MutantExecutor	TestCa	MutantIn	Class und	Verdict	TestType	Device	Date	Time	Algorithm	BuildAPKT	PushTime	InstallTim	Trun
2	183e79b6-741	mangosta-a	MS	...	1	inaka.con	A	androidTe	4bd3f236	16/5/20	4:44:47 p. m.	allAgainstAll				872
3	183e79b6-741	mangosta-a	MS	...	1	inaka.con	A	androidTe	4bd3f236	16/5/20	4:44:58 p. m.	allAgainstAll				1089
4	183e79b6-741	mangosta-a	MS	...	1	inaka.con	A	androidTe	4bd3f236	16/5/20	4:45:08 p. m.	allAgainstAll				1032
5	183e79b6-741	mangosta-a	MS	...	1	inaka.con	A	androidTe	4bd3f236	16/5/20	4:45:19 p. m.	allAgainstAll				1096
6	183e79b6-741	mangosta-a	MS	...	1	inaka.con	A	androidTe	4bd3f236	16/5/20	4:45:28 p. m.	allAgainstAll				919
7	183e79b6-741	mangosta-a	MS	...	1	inaka.con	A	androidTe	4bd3f236	16/5/20	4:45:39 p. m.	allAgainstAll				1101
8	183e79b6-741	mangosta-a	MS	...	1	inaka.con	A	androidTe	4bd3f236	16/5/20	4:45:49 p. m.	allAgainstAll				1023
9	183e79b6-741	mangosta-a	MS	...	1	inaka.con	A	androidTe	4bd3f236	16/5/20	4:46:00 p. m.	allAgainstAll				1103
10	183e79b6-741	mangosta-a	MS	...	1	inaka.con	A	androidTe	4bd3f236	16/5/20	4:46:10 p. m.	allAgainstAll				926
11	183e79b6-741	mangosta-a	MS	...	1	inaka.con	A	androidTe	4bd3f236	16/5/20	4:46:19 p. m.	allAgainstAll				930
12	183e79b6-741	mangosta-a	MS	...	1	inaka.con	A	androidTe	4bd3f236	16/5/20	4:46:27 p. m.	allAgainstAll				830
13	183e79b6-741	mangosta-a	MS	...	1	inaka.con	A	androidTe	4bd3f236	16/5/20	4:46:36 p. m.	allAgainstAll				864

Figure 14. Two excerpts of the *global.txt* file, generated by *BacterioWeb*.

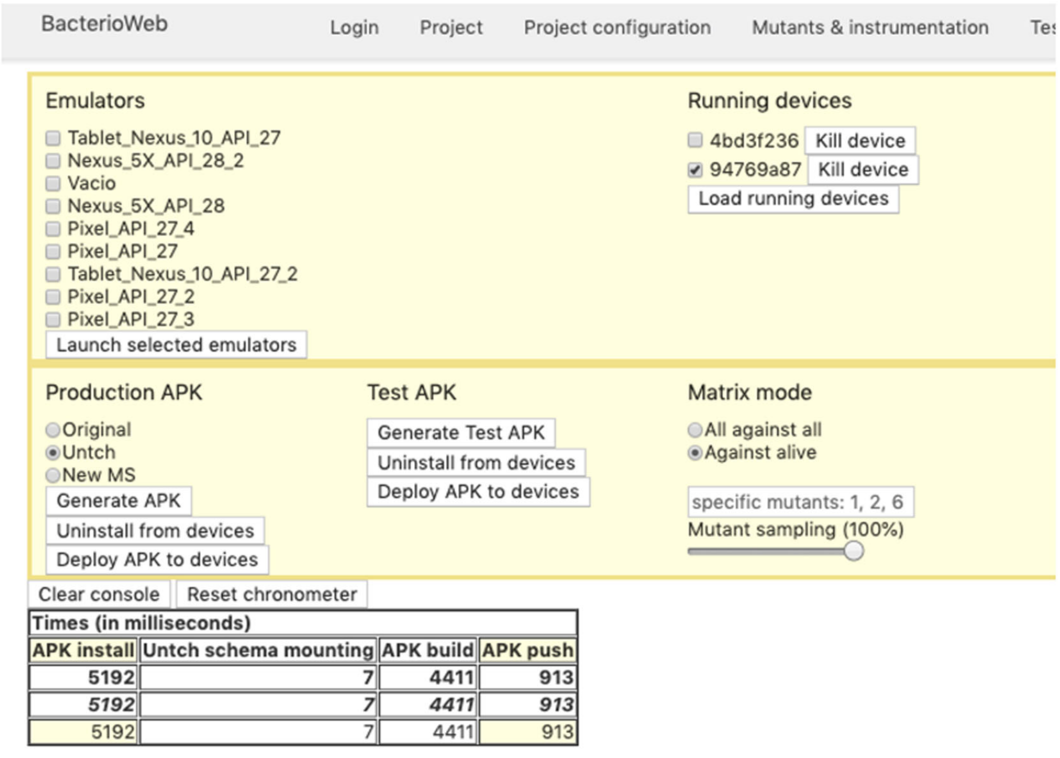

Figure 15. Summary of times in *BacterioWeb*.

© 2021 John Wiley & Sons, Ltd

DOI: 10.1002/stv

Figure 14 corresponds to two different executions of a test suite against the *Mangosta* project. The execution in the top row took place in the morning (column *K*) of May 20, 2020 (column *J*). The run times always appear on column *P*. However, the build, push and install times only appear when these tasks are effectively performed: for example, the mutant in the first row (mutant number *117,* column *E*) was deployed to the device *4bd3f236* (column I). Since the test case kills this mutant (refere to the *Verdict* in column *G*) and we are executing with *Only alive* (column *L*), the execution of the mutant is interrupted and, on row 302, the deployment of the next mutant (number 131) to the *4bd3f236* starts.

The bottom row of Figure 14 corresponds to an execution with Mutant Schema (column C). So, columns *M*, *N* and *O* are empty because the building, pushing and installing are performed before launching the test suite execution. Anyway, *BacterioWeb* shows the tester the times spent in its test execution window (Figure 15). Also note the presence of the time devoted to mounting the mutant schema.

Therefore, the times collected are the following:

- $T_{compile}$, which is the time required for compiling the application. This time is only applicable for instrumented test cases.
- T_{push}, which is the time for pushing the application from the server to the device. It is also only applicable for instrumented test cases.
- $T_{install}$, which is the time for installing the application onto the device. This time is only applicable for instrumented test cases.
- T_{run}, which is the time spent in executing one test case against one mutant.

6.7. *Experimental results*

This section is intended to provide answers to the research questions that have been raised.

- RQ1: Can we accept the assumptions made in the Epigraph 4.6 (assumptions) of Section 4 (mathematical models)? This is, are the times for compiling, pushing, installing and running, independently of the use or not of Mutant Schema?

As we have said in Section 6.6, each test suite has been executed three times in each of the eight modes shown in Figure 13. That is, each test suite has been executed 24 times in every project. These executions have produced a lot of data.

For comparing $T_{compile}$, T_{push} and $T_{install}$ with and without Mutant Schema, we have taken 3,000 random compilations, pushes and installations with No Mutant Schema and 100 compilations with Mutant Schema. The reason of the difference in the sample sizes is that No Mutant Schema compiles, pushes and installs a lot of times, while Mutant Schema does them only once.

Then, for each time, we have compared the values got and applied the Student's *T*. We have two hypotheses:

H0: $T_{compile}$, T_{push} and $T_{install}$ are the same with or without Mutant Schema.
H1: $T_{compile}$, T_{push} and $T_{install}$ are different with and without schema.

Table X. $T_{compile}$ is similar both with and without Mutant Schema.

	$T_{compile}$ (ms)						
	NoTech			MS			
Project	Mean	SD	Samples	Mean	SD	Samples	*p* value
AlarmClock	–	–	–	–	–	–	
JustSit	2,523	663	3,000	2,486	624	100	0.58
Mangosta	4,373	887	3,000	4,279	911	100	0.30
Figures	1,245	21	3,000	1,241	18	100	0.06
Dexter	1,692	101	3,000	1,684	49	100	0.43
Turismo	2,115	120	3,000	2,095	210	100	0.11
Kuar	1,983	123	3,000	2,001	138	100	0.15
WordPress	5,336	240	3,000	5,301	326	100	0,16

© 2021 John Wiley & Sons, Ltd

DOI: 10.1002/stv

If the times were different, then *H0* would be rejected (with $p\ value < 0.05$) and *H1* would be accepted. If $p\ value \geq 0.05$, then there is no evidence enough to reject *H0*.

Tables X, XI and XII respectively summarize the data collected from compiling, pushing and installing in the analysed projects. Note that all *p* values are greater than 0.05, what leads us to not reject *H0*.

Some results may surprise the reader and have surprised the authors. In particular, one expects that $T_{compile}$ should be significantly greater with Mutant Schema, since almost every call, arithmetic operation or comparison is translated into a call to a method in the *MutantDriver*.

Table XI. T_{push} is similar both with and without Mutant Schema.

	T_{push} (ms)						
	NoTech			MS			
Project	Mean	SD	Samples	Mean	SD	Samples	*p* value
AlarmClock	–	–	–	–	–	–	
JustSit	51	18	3,000	49	23	100	0.28
Mangosta	887	142	3,000	893	151	100	0.68
Figures	102	10	3,000	101	12	100	0.33
Dexter	208	85	3,000	212	106	100	0.65
Turismo	578	186	3,000	588	197	100	0.60
Kuar	281	140	3,000	278	56	100	0.83
WordPress	1,184	452	2,096	1,209	511	100	0.59

Table XII. $T_{install}$ is similar both with and without Mutant Schema.

	$T_{install}$ (ms)						
	NoTech			MS			
Project	Mean	SD	Samples	Mean	SD	Samples	*p* value
AlarmClock	–	–	–	–	–	–	
JustSit	979	603	3,000	1,012	611	100	0.59
Mangosta	5,129	189	3,000	5,097	211	100	0.10
Figures	4,431	456	3,000	4,344	480	100	0.06
Dexter	4,540	170	3,000	4,509	206	100	0.08
Turismo	6,185	141	3,000	6,191	143	100	0.68
Kuar	4,798	2,711	3,000	4,526	2,634	100	0.32
WordPress	10,132	2,911	3,000	10,159	10,318	3,001	0.53

Table XIII. T_{run} is similar both with and without Mutant Schema.

	T_{run} (ms)					
	NoTech		MS			
Project	Mean	SD	Mean	SD	Sample	*p* value
AlarmClock	2,528	801	2,489	815	3,000	0.06
JustSit	9,310	3,129	9,397	3,213	3,000	0.15
Mangosta	10,239	3,755	10,242	3,978	3,000	0.98
Figures	7,453	464	7,465	464	3,000	0.32
Dexter	4,047	666	4,051	689	3,000	0.82
Turismo	11,552	2,960	11,486	2,888	3,000	0.38
Kuar	15,432	8,146	15,704	7,870	3,000	0.19

© 2021 John Wiley & Sons, Ltd

DOI: 10.1002/stv

With respect to T_{run}, which is the execution time of a tests case against a mutant, the sample size in both cases is 3,000. This is because we save (remind the *global.txt* file in Figure 14) the execution time of every test case against every mutant and in both cases (with and without Mutant Schema) thousands of executions have been run in each test cycle. Thus, in this case, we can compare samples of the same size. As it is seen in Table XIII, neither the null hypothesis can be rejected: that is, we cannot distinguish whether a test case execution against a mutant has been executed with or without Mutant Schema.

6.7.1. Partial conclusions. Since there is no evidence to reject *H0*, we will assume for the remaining experiment that compiling, pushing and installing an app onto a device is the same with independence of the use of Mutant Schema.

Note that the veracity of this assumption would allow us to build mathematical models with almost not executing tests.

- RQ2: How good are the mathematical models to predict the mutation testing time?

From the huge amount of data collected by *BacterioWeb*, we have built several tables for each project. As an example, next tables summarize the results for the Mangosta project. Mangosta has a test suite with 31 test cases, and *BacterioWeb* generates 1,579 mutants for it.

The first five columns in each row includes the number of devices, a number of mutants, the reached mutation score and the mean of the measured total run time. Last four columns show the execution time (which is the run time plus the time for compiling, pushing and installing): the *Actual* column is the time actually measured, and *Estimated* is the time calculated according to the Mathematical models.

We have executed three times all the test cases against the mutants, in the eight variants: for Mangosta, for example (Tables XIV–XVII), we have executed three times the 31 test cases against the 1,579 mutants using 1 and 2 devices, *Mutant Schema (MS)* and *No Mutant Schema (NoMS)*, *All against all (AA)* and *Only alive (OA)*. The number of mutants in each row has been randomly selected from the 1,579, being the same mutants for each variant.

Figure 16 depicts the data about actual and estimated times shown in the previous tables for 1 device. As it is seen, the adjustment of the measured and estimated curves is almost perfect.

It is worth noting that we get similar results in all the analysed projects.

Figure 17 unifies the actual execution times of the tables proceeding from the Mangosta project, which in turn proceeds from the technique's combinations listed in Figure 13. As expected, the

Table XIV. Times with the *Mangosta* project, without Mutant Schema and All against all.

Mangosta, No Mutant Schema, All against all						
					T_{exec} (h)	
Devices	|M|	M.Score	Executions	Total T_{run} (h)	Actual	Estimated
1	60	0.98	1860	5.4	5.6	5.5
1	100	0.98	3,100	9.0	9.3	9.1
1	300	0.99	9,300	27.1	28.0	27.3
1	600	1	18,600	54.3	56.0	54.6
1	900	0.99	27,900	81.4	84.0	81.9
1	1,200	0.99	37,200	108.5	112.0	109.3
1	1,579	0.99	48,949	142.8	147.4	143.8
2	60	0.98	1860	2.7	2.8	2.7
2	100	0.98	3,100	4.5	4.7	4.6
2	300	0.99	9,300	13.6	14.0	13.7
2	600	1	18,600	27.1	28.0	27.3
2	900	0.99	27,900	40.7	42.0	41.0
2	1,200	0.99	37,200	54.3	56.0	54.6
2	1,579	0.99	48,949	71.4	73.7	71.9

© 2021 John Wiley & Sons, Ltd

DOI: 10.1002/stv

Table XV. Times with the *Mangosta* project, without Mutant Schema and Only Alive.

Mangosta, No Mutant Schema, Only alive						
					T_{exec} (h)	
Devices	|M|	M.Score	Executions	Total T_{run} (h)	Actual	Estimated
1	60	0.98	1,623	4.6	4.8	4.8
1	100	0.98	2,702	7.6	7.9	8.0
1	300	0.99	7,861	23.1	24.0	23.2
1	600	1	16,441	44.6	46.3	48.5
1	900	0.99	24,107	70.7	73.3	71.2
1	1,200	0.99	32,295	93.8	97.3	95.3
1	1,579	0.99	43,250	124.1	128.7	127.6
2	60	0.98	1,623	2.3	2.4	2.4
2	100	0.98	2,702	3.7	3.9	4.0
2	300	0.99	7,861	11.7	12.2	11.6
2	600	1	16,441	21.8	22.7	24.2
2	900	0.99	24,107	35.2	36.5	35.6
2	1,200	0.99	32,295	46.1	47.8	47.7
2	1,579	0.99	43,250	63.0	65.3	63.8

Table XVI. Times with the *Mangosta* project, with Mutant Schema and All against all.

Mangosta, Mutant Schema, All against all						
					T_{exec} (h)	
Devices	|M|	M.Score	Executions	Total T_{run} (h)	Actual	Estimated
1	60	0.98	1860	5.4	5.4	5.3
1	100	0.98	3,100	8.9	9.2	8.8
1	300	0.99	9,300	26.9	27.7	26.5
1	600	1	18,600	54.4	56.1	52.9
1	900	0.99	27,900	78.3	80.9	79.4
1	1,200	0.99	37,200	108.5	111.9	105.8
1	1,579	0.99	48,949	141.8	146.3	139.2
2	60	0.98	1860	2.6	2.6	2.6
2	100	0.98	3,100	4.3	4.3	4.4
2	300	0.99	9,300	13.3	13.3	13.2
2	600	1	18,600	27.7	27.7	26.5
2	900	0.99	27,900	38.6	38.6	39.7
2	1,200	0.99	37,200	54.0	54.0	52.9
2	1,579	0.99	48,949	68.9	68.9	69.6

fastest combination is Mutant Schema, Only Alive and two devices *(MS*, *OA* and *2 devices)*. The eight combinations were executed for all the projects, getting similar results in all the cases.

6.7.2. Partial conclusions. We can conclude that the estimated times from our mathematical models and the real times obtained in the execution of test cases are very similar. Therefore, we can estimate a priori how long it may take to execute mutation tests in a mobile application and, depending on its feasibility, the tester can make decisions about which combination of cost reduction techniques is more convenient. The experimental data have shown that: (1) the estimated and actual times are very similar, (2) the most efficient combination of the applied cost reduction techniques is Mutant Schema, Only Alive and Parallel Execution and (3) Parallel Execution is the most cost-savings technique.

- RQ3: How does the number of mutants influence on the improvement factor?

© 2021 John Wiley & Sons, Ltd

DOI: 10.1002/stv

Table XVII. Times with the *Mangosta* project, with Mutant Schema and Only Alive.

Mangosta, Mutant Schema, Only alive						
					T_{exec} (h)	
Devices	|M|	M.Score	Executions	Total T_{run} (h)	Actual	Estimated
1	60	0.98	1,608	4.5	4.7	4.7
1	100	0.98	2,658	7.5	7.5	7.6
1	300	0.99	7,943	22.2	22.2	22.6
1	600	1	15,996	43.8	43.8	45.5
1	900	0.99	24,595	68.2	68.2	70.0
1	1,200	0.99	32,412	89.3	89.3	92.2
1	1,579	0.99	43,025	122.4	122.4	122.4
2	60	0.98	1,608	2.3	2.3	2.3
2	100	0.98	2,752	3.8	3.8	3.9
2	300	0.99	7,851	11.2	11.2	11.2
2	600	1	16,407	22.5	22.5	23.3
2	900	0.99	23,548	32.0	32.0	33.5
2	1,200	0.99	31,974	43.7	43.7	45.5
2	1,579	0.99	43,510	62.2	62.2	61.9

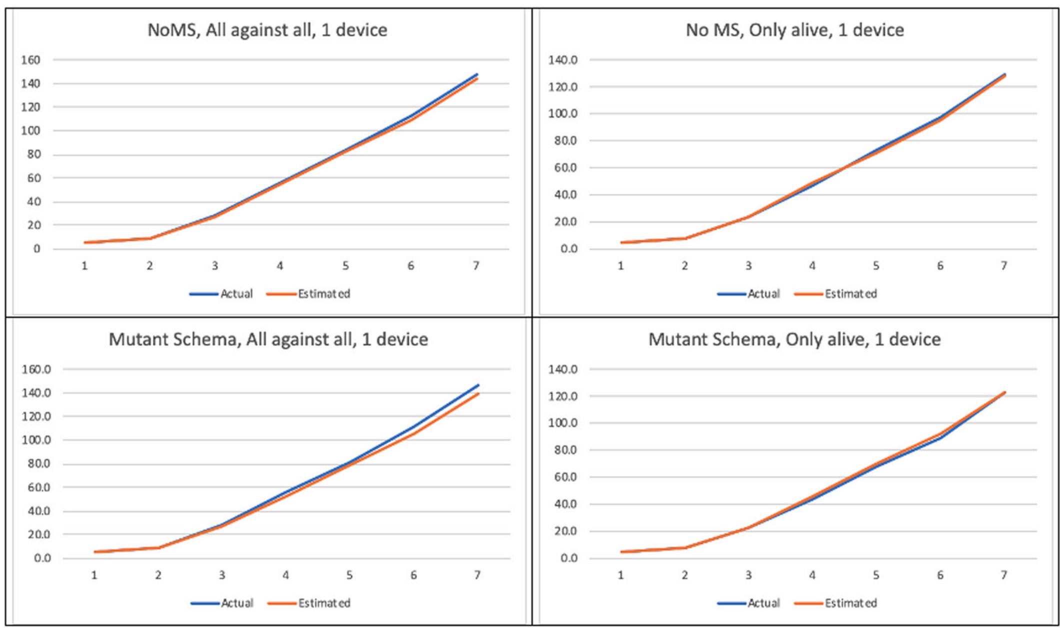

Figure 16. Actual and estimated times in the Mangosta project.

In Section 4.5, we defined *IF*, the Improvement Factor, as the quotient between the time of non-using any technique (i.e. No Mutant Schema and All against all) with the time of using one or more cost-reduction techniques:

$$IF = \frac{|M| \cdot \left(T_{compile} + T_{push} + T_{install} + \rho \cdot T_{run}\right)}{n \cdot \left(T_{compile} + T_{push} + T_{install}\right) + |M| \cdot \rho \cdot T_{run}} \tag{11}$$

We also concluded that the improvement factor got by any combination of the analysed techniques tends to stabilize when the number of mutants grows up. Since the assumptions made in the

© 2021 John Wiley & Sons, Ltd

DOI: 10.1002/stv

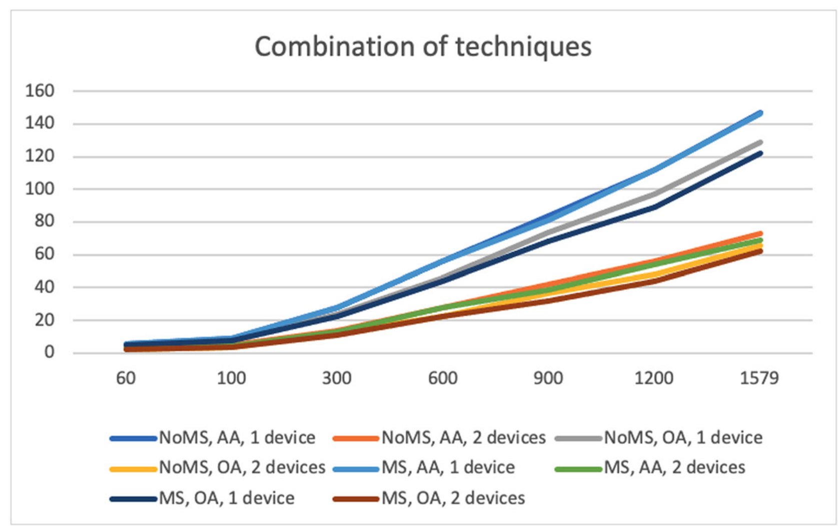

Figure 17. Results of all technique combinations in the Mangosta project.

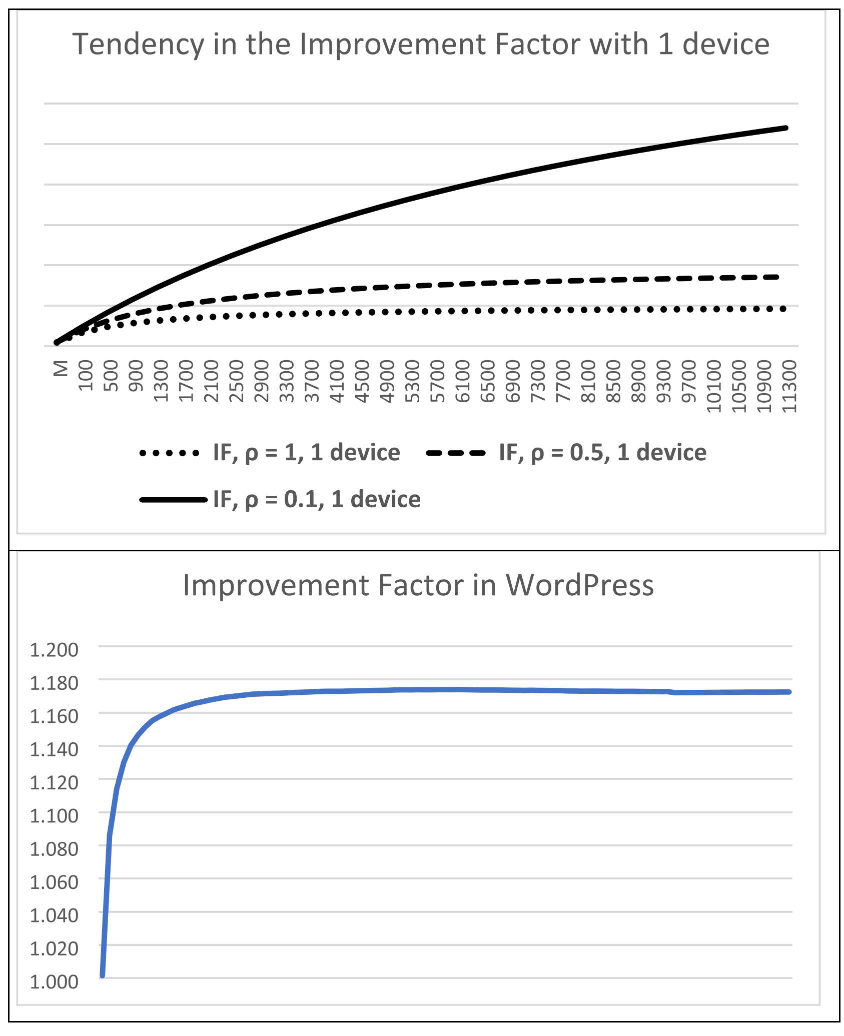

Figure 18. Theoretical tendency in the Improvement Factor with 1 device and different values of ρ (top) and tendency observed in WordPress.

© 2021 John Wiley & Sons, Ltd

DOI: 10.1002/stv

mathematical models are acceptable, we can draw the tendencies with arbitrary values for the different variables involved in Equation 11. Top side of Figure 18 shows the tendency of *IF* with different values of ρ and when the number of mutants (horizontal axis) grows up:

- The dotted line evidences that the benefit of using Mutant Schema with *All against all* ($\rho = 1$) is low if the number of mutants is high.
- The scatted line shows the tendency in *IF* with $\rho = 0.5$: this is, tests are executed with *Only alive* and there is a 'medium good luck' in the execution order of test cases.
- Finally, the solid line is the tendency with $\rho = 0.1$. This situation could correspond either to a 'very good luck' in the execution order of test cases or, much better, to a 'smart' execution algorithm that prioritizes the test cases with more ability to kill mutants. We will recall this point in the Future works section.

The actual tendency observed in WordPress (as in all the projects) is shown in the bottom side of the figure.

6.7.3. Partial conclusions. As it was predicted in the mathematical model, the experimental data show that Mutant Schema always improves the execution time: the improvement factor rises quickly when the number of mutants is low, but it stabilizes and tends to a constant from a certain number of mutants.

6.8. Threats to validity

The nature of the experiments introduces some threats to validity, which must be considered in order to evaluate the conclusions.

Construct validity is the degree to which independent and dependent variables are accurately measured [47]. All our independent variables are nominal (*presence* or *absence of: Mutant Schema, Parallel Execution* and *Only Alive*), and the dependent variable (*time*) is measured objectively by the mutation tool. To alleviate bias, we performed several repetitions of each execution so as to reduce the threat.

Internal validity is the degree of confidence in a cause–effect relationship between the factor of interest and the observed results [47]. All the variables have been controlled during the experiments in order to minimize threats to internal validity.

External validity is the extent to which the research results can be generalized to the population under study and other research settings [47]. Obviously, it is quite risky to affirm that our models are valid and applicable to any other application and environment. In order to alleviate this threat, we have used a diverse set of apps with different characteristics and a variety of mutation operators.

We consider that the most significant finding that will allow the greatest generalization of the mathematical model to other applications, contexts and environments (not only mobile mutation testing) is the one which responds to RQ1: this is, the compilation, pushing and installing times are quite similar independently of the use or not of Mutant Schema.

With respect to the execution algorithms (*All against all* and *Only against alive)*, it is important to note that, in our case, they are completely deterministic and do not implement any technique to prioritize the execution order of test cases. But, once more, the answers to RQ1, which validate the goodness of the assumptions made in the Mathematical Models section, allow to realize theoretical models that should be experimentally validated.

7. LESSONS LEARNED

Due to the number of applications, mutants, test cases, combinations of techniques and—in order to get reliable measures—the number of repetitions of each task, the process of experimentation and data collection has been very long. From the observation of *BacterioWeb* during test execution, from how it fills-in the killing matrixes and from the many result analysis made, we have recovered forgotten lessons and also learned new ones. We believe it is important to share both with the research community and with practitioners.

© 2021 John Wiley & Sons, Ltd

DOI: 10.1002/stv

```
public void test1() {
    SUT sut = ...;
    Database db = sut.getDatabase();
    db.removeAllCustomers();

    db.insertCustomer("John", "Smith");

    assertTrue(db.gestCustomers()==1);
}
```

```
public void test2() {
    SUT sut = ...;
    Database db = sut.getDatabase();

    try {
        db.deleteCustomer("John", "Smith");
        assertTrue(db.gestCustomers()==0);
    } catch (Exception e) {
        fail("Customer not found");
    }
}
```

Figure 19. Two test cases in a supposed test suite.

7.1. Test suite reduction

The reduction of the test suite size cannot make sense if the SUT's state depends on the test cases previously executed: suppose the test suite shown in Figure 19, which contains two test cases *test1* and *test2* that respectively insert and delete a customer from a database.

If the mutants that *test1* kills are all included in those killed by *test2*, *test2* is selected for the reduced test suite. However, when *test2* is executed in isolation, the test case will always reach the *fail* statement, since the customer inserted in *test1* (that is no longer executed) will not be contained in the database.

We have observed this situation, for example, in *Mangosta*, the project we are using as running example:

- Figure 20a shows a fragment of the killing matrix with 27 of its 31 test cases and some randomly selected mutants. The mutation score is 0.75. Figure 20c shows the summary of the execution: three test cases compose the selected reduced test suite:

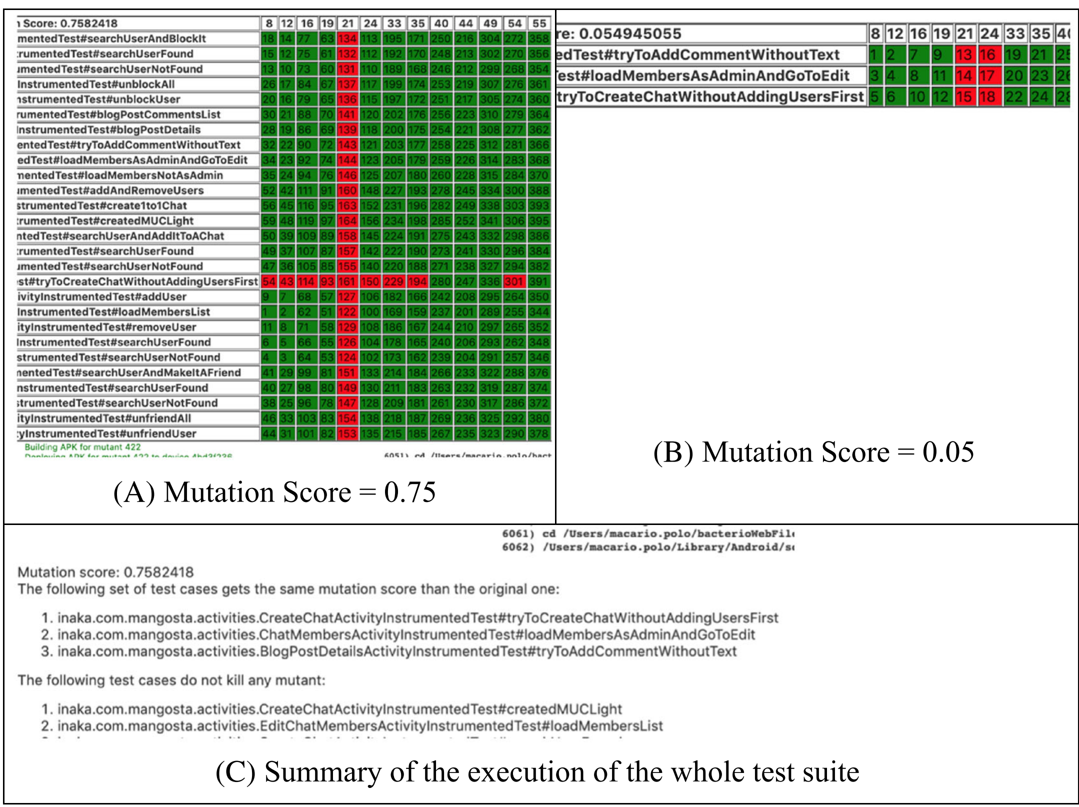

Figure 20. Reducing the test suite does not always produce reliable results.

© 2021 John Wiley & Sons, Ltd

DOI: 10.1002/stv

tryToCreateChatWithoutAddingUsersFirst, *loadMembersAsAdminAndGoToEdit* and *tryToAddCommentWithoutText*.

- Figure 20b shows some cells of the killing matrix after executing only the reduced test suite. The results are quite different:
  - With the whole test suite, the first test case kills mutants 8, 12, 16, 19, 33 and 35. This test is also the only that kills mutant 24.
  - Executing only the reduced test suite, the same test leaves those mutants alive. Moreover, the three test cases kill mutant 24. Note moreover the quite different mutation score of this supposedly equivalent test suite, that is only 0.05 (top-left cell of the killing matrix).

In order to successfully reduce a set of test cases, and for the reduced test suite to kill the same mutants and obtain the same mutation score as the original test suite, it is imperative that each test case be repeatable, autonomous and independent of the other cases in the test suite.

7.2. *Test case prioritization*

1. In some projects, there are test cases whose execution takes much more time than others. The 14 test cases of *AlarmClock*, for example, are grouped in the three files appearing in Table XVIII, that also shows the mean execution time of each test case. As it is seen, the last test case (*snoozeAlarm …)* needs more than 1 min, what slows down the overall execution time very much. Since the mutation process must only start after all the test cases do not find any error in the original system, it is a good idea to organize the execution against mutants in groups of test cases, sorting them by the expected execution time before launching the tests or, even, excluding the longest test cases from the test suite.

The mutation testing processes of Offut [6] and of Usaola and Mateo [7] include, as an essential technique to reduce costs, the execution of tests only against the mutants remaining alive (what we have called *Only Alive*). Figure 21 redefines our process [7] with the consideration of the execution time (specially worrying in mobile testing) as a mechanism technique for cost reduction: the tester starts executing the test suite *T* against the SUT, *S*. If there are no errors, s/he separates (node 3) the test cases in several test files ($TF1 \dots TF_n$) according to the test case execution times. If it is the first execution, mutants must be generated (node 4) and the test files iteratively launched against the mutants, removing the killed mutants (node 6) and analysing the mutation score: if the prefixed threshold is reached, the process can stop. Otherwise, if there are more test files, the tester launches the next one against the mutants; if there are no more test files, s/he must create a new one to visit and try to kill the mutants remaining alive (node 7). This new test file is executed against the original system on (node 8): if it finds any error, the system must be fixed (and new mutants will have to be later generated because *S* has changed); otherwise, the new file can be directly executed against *M*.

Table XVIII. Mean execution times of Alarm Clock's test cases.

Test file	Test case	Mean time (ms)
DaysOfWeekTest	testSaturday …	1,900
	testMondayA …	1,932
	testSunday …	2,095
DurationUtilsTest	testBreakdown	2,139
AlarmTest	setRecurringDays …	2,014
	snoozeAlarm …	2,170
	alarm_RingsAt_... (1)	2,205
	alarm_RingsAt_... (2)	2,220
	alarm_RingsAt_... (3)	2,273
	alarm_RingsAt_... (4)	2,293
	alarm_RingsAt_... (5)	2,418
	alarm_RingsAt_... (6)	2,443
	snoozeAlarm_IsSnoozed_...	62,172

© 2021 John Wiley & Sons, Ltd

DOI: 10.1002/stv

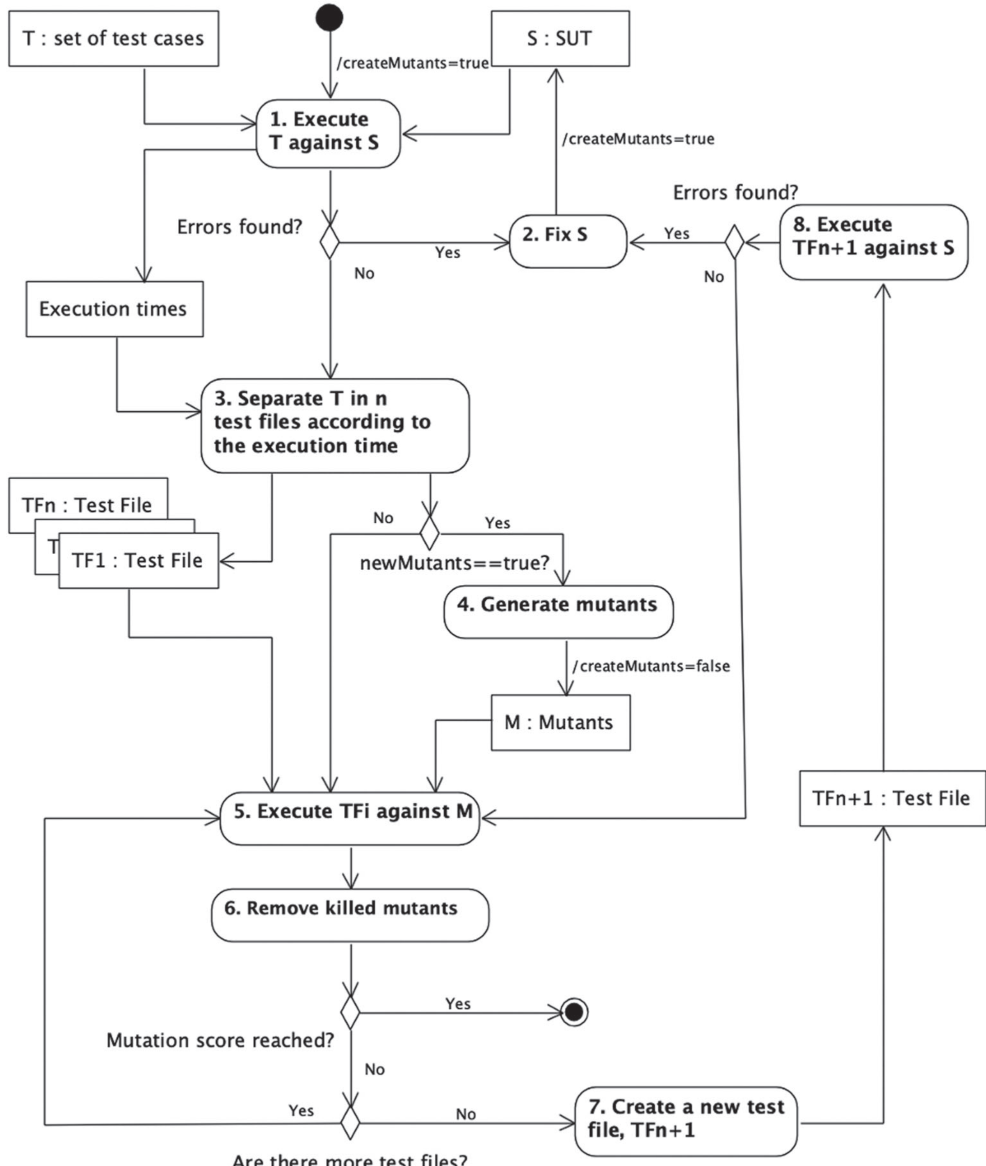

Figure 21. A mutation process, specifically adapted to mobile software.

For the case of the *AlarmClock* project, excluding the longest test case from the first execution saves 4 s in the execution of every mutant. In the second iteration (node 5), this test case will be launched only against the mutants remaining alive, which are far fewer than before. The risk, with this approach, is that test cases in the first test files kill a small number of mutants.

2 All test cases under the *androidTest* folder (Figure 2) require the generation of an *apk* file and its installation onto a device. Some testers leave the unit tests in this folder, what slows down the execution time. It is important to leave the unit tests in its own folder, since they are executed much more quickly.

3 To avoid the execution of test cases against mutants that will not be visited, the tester should not generate mutants with Android-specific operators for executing unit tests.

4 When *BacterioWeb* fills the killing matrix, it shows a number that indicates the order in which the test case has been executed against every mutant (Figure 23). This information is interesting for sorting the test suite when facing future regression test cycles.

© 2021 John Wiley & Sons, Ltd

5 The process described in Figure 21 can be adapted for regression testing: suppose a system *S* composed by classes *A, B* and *C*. Let be *TS* a test suite that (1) does not find any fault in *S*, (2) is mutation-adequate and (3) only contains the best test cases obtained from the application of a test suite reduction algorithm (Section 3.2). If one the classes in the system (let be *A*) changes after the addition of, for example, a new functionality, the tester must, in the first time, to re-execute *TS* against the new version of *S* (let be *S′)* to find possible new faults. If *TS* does find no faults, then it is recommendable to generate new mutants (i.e. *A1* and *A2*) only for the classes that have changed and re-execute *TS* only against these new mutants. According to the figure, if *TS* does not reach the mutation score threshold, new test cases should be added to TS.

7.3. *Detection of equivalent mutants*

1 Although Mutant Schema improves the total execution time, it makes it much more difficult to find the reason why a mutant remains alive due to a poorer legibility of the code (Figure 9, Section 5.2).

7.4. *Structure of test cases*

1 Some mutants may lead the app to enter in an infinite loop (e.g. if a counter variable is decreased inside a loop). Android test cases can be annotated with a timeout label: if the test case has not finished after this timeout, the mutant is considered killed. Some authors resolve this with weak mutation by the instrumentation of the code.
2 It is also interesting to include frequent assertions in test cases (i.e. not only an oracle at the end of the test case): when we started to test *Kuar*, every test case reproduced a complete match. This required performing many movements to drive the board to its final state (from the left side of Figure 22 to the right side) and took a long time because initially there was only one oracle instruction (*assertX)* at the end of each test case to check the final result. In order to detect killed mutants as soon as possible, we introduced frequent assertions (one *assert* after each movement). This considerably accelerates test execution.

8. FUTURE WORK

Besides the convenience of applying cost-reduction techniques (both those analysed in this article as other ones, such as the reduction of the number of mutants), mutation testing for mobile software is quite costly, and it requires researching on new techniques and techniques combinations that,

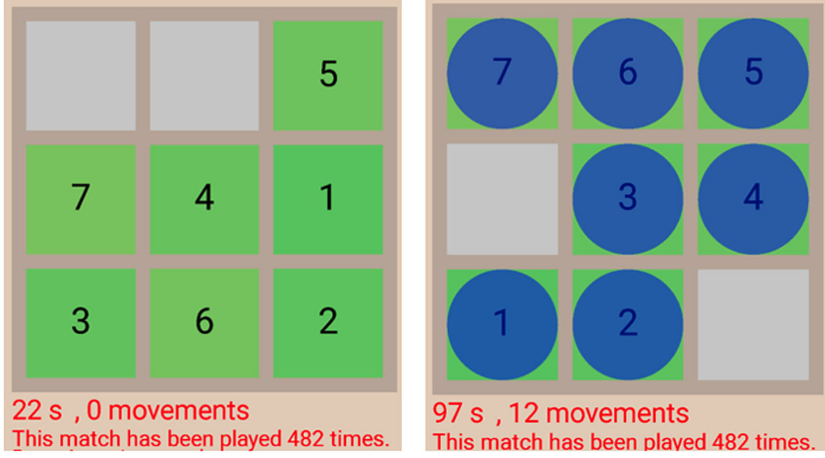

Figure 22. Kuar’s board.

© 2021 John Wiley & Sons, Ltd
DOI: 10.1002/stv

furthermore, could be extended to other kinds of systems. Below, we describe some lines of work we consider quite interesting and that could drive future research.

8.1. Specific operators for mobile software and operators subsumption

In this article, we have applied classical and Android-specific mutation operators. Some of these have been reproduced from the descriptions given in the literature [4, 5], and we have implemented some others. The idea of mutation operators specifically built for a concrete technology is to introduce typical errors of such technology.

Many of the Android-specific operators introduce errors that may be also inserted by classic operators. Consider for example the MDL operator that deletes a lifecycle method of an activity. Deng *et al*. [4] propose this operator, but they warn that it is 'similar to the Overriding Method Deletion in muJava' [33]. Thus, if the tool offers the tester both operators, two redundant mutants will be generated.

Another example is IPR, which replaces the second parameter of the *putExtra* method by a default value (zero if it is primitive, *null* and the empty string if it is a String, etc.). MJP is similar to IPR, but changes the values put in JSON objects. Actually, the same mutants can be generated with other classical operators, such as the Scalar Variable Replacement operator of the classic Mothra system [48].

Therefore, it is likely required to carry out an extensive study of operators' subsumption, in order to avoid the generation of duplicate mutants.

8.2. Mutant generation guided by metrics

There are many studies that correlate software metrics with the fault-proneness of the system's modules [49–50]. With a previous static analysis of the system, the tester could focus mutant generation on those classes that have more coupling, which is the best predictor according to those studies.

As an example, the *Figures* project (that was specifically developed for testing some of the *BacterioWeb* characteristics) has the *LocalCalculus* and *RemoteCalculus* classes, which are used to determine the type and perimeter of the figure in the self app or via a query to a web server. Since almost any test scenario runs one of these two classes, it can be promising to determine 'execution clusters' to concentrate the mutant generation on them.

8.3. Mutant execution guided by static analysis

Such 'execution clusters' would avoid the execution of test cases against mutants that they will not kill: for example, it probably makes no sense to execute a test case that determines the type of a *Triangle* against a mutant of *Quadrilateral*. Before the execution of the tests, a static analysis of the code could help to relate test cases with classes of the SUT, therefore producing a more fine-grained set of clusters, composed now by tests and classes of the SUT. The result of this analysis would guide the execution of each test only against the mutants from the classes it will visit.

Some authors have researched on mutant clustering to reduce the execution time, but applying other strategies: Ma and Kim [51] and Yu and Ma [52] built the clusters based on the mutants that are expected to produce the same result with test cases. Ji *et al*. [53] proposed to cluster mutants based on their Hamming distance.

8.4. Algorithms for Parallel Execution

At first glance, putting more devices to execute tests and mutants is a brute force mechanism that, undoubtedly, must improve the overall execution time.

As a mean, we can guess that using 2 devices will require half time than using 1, and that using n will take $1/n$. We have observed that his premise is true when there are many mutants (suppose 1,000 mutants, 100 of which require more time than the others: these 100 mutants will be fairly distributed to the two devices). However, when there are few mutants, there may be significant differences among the devices.

© 2021 John Wiley & Sons, Ltd

DOI: 10.1002/stv

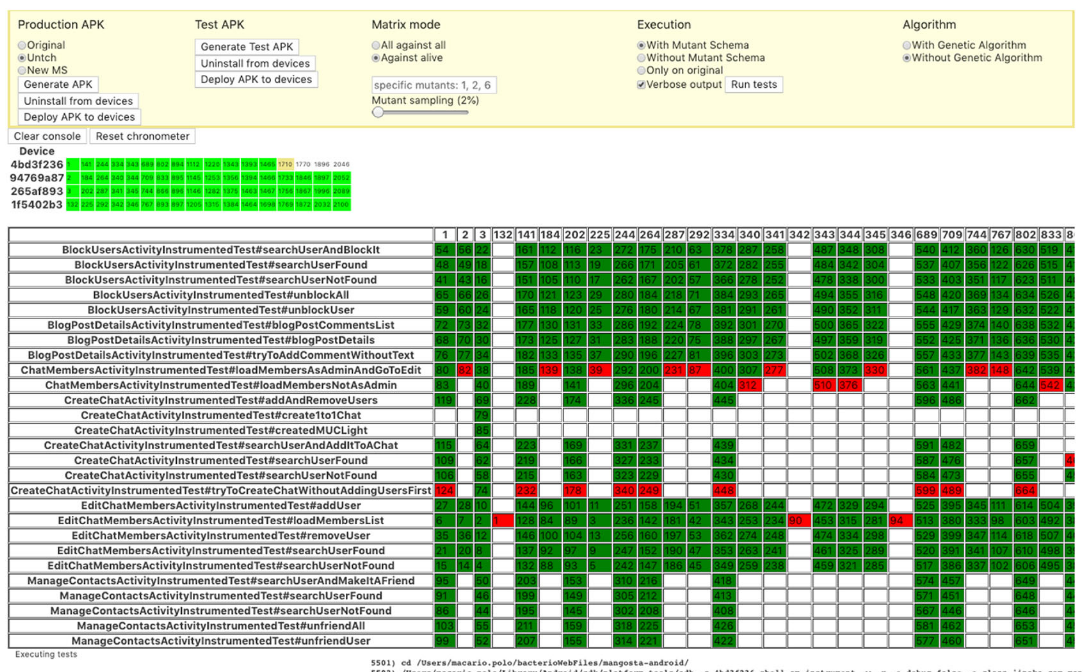

Figure 23. Killing matrix during one execution.

Figure 23 shows an example of *BacterioWeb* executing the *Mangosta* test cases against a small sample (only 2%, since the figure is only for illustrating purposes) of mutants. As it can be seen, it is applying the *Only against alive* algorithm (refer to the figure the selected 'Matrix mode' radio button) and *With Mutant Schema*. There are four devices (Samsung tablets, model SM-T590, Android 8.1.0 and 3 Gb RAM) that have received 17 mutants each. The second, third and fourth devices have finished the execution, while the first one still has four mutants left.

A more equitable time distribution could be achieved by distributing the mutants with other parallel execution algorithms. For example, when a device finishes the execution of the test suite (either with *Only against alive* or with *All against all*), it could ask for a new mutant to be executed.

In a previous article, we analysed five different algorithms for parallel execution in mutation testing [32]: *Distribute mutants between operators*, *Distribute test cases*, *Give mutants on demand*, *Give test cases on demand* and *Parallel execution with dynamic ranking and order*.

It is possible to build new mathematical models or to extend those in Section 5, with the inclusion of these or of other algorithms.

9. CONCLUSIONS

Several recent researchers have analysed the suitability of mutation for testing mobile software. Most of these works focus on the proposal and application of specific mutation operators to reproduce the particular errors of this kind of software. Even though they highlight the suitability of mutation for this context, some of them also emphasize the great amount of time it requires.

In order to extend the research lines in this area, this paper has analysed how different classic cost reduction techniques influence on the mutation testing time of mobile software.

The main contributions of this article are related to how several well-known cost reduction techniques help to the effective improvement of testing time. The techniques we have used are *Mutant Schema*, *Only Alive* and *Parallel Execution*, as well as several combinations of them. The baseline for the comparisons is a classic model of mutation testing, where there is a complete cycle of compilation, deployment and test execution per mutant.

© 2021 John Wiley & Sons, Ltd

The first technique considered is *Mutant Schema*, which requires to generate the schema and just one deployment onto the running device. Being the deployment a very significant task in mobile testing, the obtained results always show meaningful cost savings when Mutant Schema is applied. However, an interesting finding is that its improvement factor tends to be asymptotic with respect to the number of mutants: in fact, as more mutants there are and greater is the execution time, less significant is the influence of the deployment time on the total cost.

The second technique used compares the execution of the test cases against all the mutants versus *only the mutants remaining alive*. Here, the improvement in the testing time depends on the quality of the test cases: as earlier the mutants are killed, less test cases will have to be executed.

The third technique is Parallel execution, which has evidenced to be the most influencing cost-reduction factor. The experiments have shown that the reduction in time is roughly proportional to the number of devices. The total execution time is the time required by the device that needs more time for executing its set of mutants. In this factor influence both the characteristics of the device (memory, processor...) as the nature of every mutant. When the number of mutants is very high, they are fairly distributed on the devices (both the ‘quick’ and the ‘slow’ mutants). Thus, the reduction is not strictly $1/n$, but it is very approximate. Even though, the reduction may be improved with other parallel execution algorithms (Section 8.4).

In our opinion, this paper complements other research works on mutation testing applied to mobile software. One additional contribution of this article is the suitability of the proposed mathematical models for describing the execution time of test cases in mutation testing. This result is interesting for build prediction models before implementing tools and techniques for mutation testing, what can shorten the research times.

ACKNOWLEDGEMENTS

This work has been partially funded by the BIZDEVOPS-Global project (RTI2018-098309-B-C31), Ministerio de Economía, Industria y Competitividad (MINECO) & Fondo Europeo de Desarrollo Regional (FEDER), and the TESTIMO project (Consejería de Educación, Cultura y Deportes de la Junta de Comunidades de Castilla La Mancha, y Fondo Europeo de Desarrollo Regional FEDER, SBPLY/17/180501/000503). Isyed Rodríguez has a scholarship from the Chilean Agencia Nacional de Investigación y Desarrollo, ANID (ANID-PCHA/21160055). The authors would like to thank Mario Piattini and Diego Soler for their valuable suggestions on a previous version of this paper.

REFERENCES

1. Jia Y, Harman M. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering* 2011; **37**(5):649–678. https://doi.org/10.1109/TSE.2010.62
2. Ferrari FC, Pizzoleto AV, Offutt J. A systematic review of cost reduction techniques for mutation testing: preliminary results, In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops*, pp. 1–10, 2018. https://doi.org/10.1109/ICSTW.2018.00021
3. Untch RH, Offutt AJ, Harrold MJ. Mutation analysis using mutant schemata. *SIGSOFT Software Engineering Notes* 1993; **18**(3):139–148. https://doi.org/10.1145/174146.154265
4. Deng L, Offutt J, Ammann P, Mirzaei N. Mutation operators for testing Android apps. *Information and Software Technology* 2017; **81**:154–168. https://doi.org/10.1016/j.infsof.2016.04.012
5. Escobar-Velasquez C, Linares-Vasquez M, Bavota G, Tufano M, Moran KP, di Penta M, Vendome C, Bernal-Cardenas C, Poshyvanyk D. Enabling mutant generation for open- and closed-source Android apps. *IEEE Transactions on Software Engineering* Apr. 2020; 1–1. https://doi.org/10.1109/tse.2020.2982638
6. Offutt AJ. A practical system for mutation testing: help for the common programmer, in *Proc. IEEE Int. Test Conf. TEST Next 25 Years*, 824–830, 1994. https://doi.org/10.1109/TEST.1994.528535
7. Usaola MP, Mateo PR. Mutation testing cost reduction techniques: a survey. *IEEE Software* 2010; **27**(3):80–86. https://doi.org/10.1109/MS.2010.79
8. “Robolectric.” http://robolectric.org [accessed May 10, 2020].
9. Kirubakaran B, Karthikeyani V. Mobile application testing—challenges and solution approach through automation. In *2013 International Conference on Pattern Recognition, Informatics and Mobile Engineering*. IEEE: Salem, India, 2013; 79–84. https://doi.org/10.1109/ICPRIME.2013.6496451

© 2021 John Wiley & Sons, Ltd

10. Kim H, Choi B, Wong WE. Performance testing of mobile applications at the unit test level, in *2009 Third IEEE International Conference on Secure Software Integration and Reliability Improvement*, 2009, pp. 171–180. https://doi.org/10.1109/SSIRI.2009.28
11. Muccini H, di Francesco A, Esposito P, Di Francesco A, Esposito P. Software testing of mobile applications: challenges and future research directions. In *2012 7th International Workshop on Automation of Software Test (AST)*, IEEE: Zurich, Switzerland, 2012; 29–35. https://doi.org/10.1109/IWAST.2012.6228987
12. Offutt AJ. Investigation of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodology* 1992; **1**(1):3–18.
13. Jabbarvand R, Malek S. µDroid: an energy-aware mutation testing framework for Android, in *Proc. 2017 11th Jt. Meet. Found. Softw. Eng. - ESEC/FSE 2017*, pp. 208–219, 2017. https://doi.org/10.1145/3106237.3106244
14. 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, in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering ACM*, pp. 233–244, 2017. https://doi.org/10.1145/3106237.3106275
15. Paiva ACR, Gouveia JMEP, Elizabeth JD, Delamaro ME. Testing when mobile apps go to background and come back to foreground, *Proceedings of the 2019 IEEE 12th International Conference on Software Testing, Verification, Validation Workshop. ICSTW 2019*, pp. 102–111, 2019. https://doi.org/10.1109/ICSTW.2019.00038
16. Morgado IC, Paiva ACR. Impact of execution modes on finding android failures. *Procedia Computer Science* 2016; **83**:284–291. https://doi.org/10.1016/j.procs.2016.04.127
17. Jia Y, Harman M. Constructing subtle faults using higher order mutation testing, in *Proceedings of the 8th IEEE International Working Conference on Source Code Analysis and Manipulation SCAM 2008*, pp. 249–258, 2008. https://doi.org/10.1109/SCAM.2008.36
18. Polo M, Piattini M, Garía-Rodríguez I. Decreasing the cost of mutation testing with second-order mutants. *Software Testing, Verification & Reliability* 2009; **19**(2):111–131. https://doi.org/10.1002/stvr.392
19. Mateo PR, Usaola MP, Alemán JLF. Validating 2nd-order mutation at system level. *EEE Transactions on Software Engineering* 2013; **39**(4):570–587. https://doi.org/10.1109/TSE.2012.39
20. Langdon WB, Harman M, Jia Y. Efficient multi-objective higher order mutation testing with genetic programming. *Journal of Systems and Software* 2010; **83**(12):2416–2430. https://doi.org/10.1016/j.jss.2010.07.027
21. Abuljadayel A, Wedyan F. An approach for the generation of higher order mutants using genetic algorithms. *International Journal of Intelligent Systems and Applications* 2018; **11**(1):34–45. https://doi.org/10.5815/ijisa.2018.01.05
22. Wong WE, Mathur AP. Reducing the cost of mutation testing: an empirical study. *Journal of Systems and Software* 1995; **31**(3):185–196. https://doi.org/10.1016/0164-1212
23. Derezinska A, Rudnik M. Evaluation of mutant sampling criteria in object-oriented mutation testing, in *2017 Federated Conference on Computer Science and Information Systems (FedCSIS)*, vol. **11**, pp. 1315–1324, 2017. https://doi.org/10.15439/2017F375
24. Offutt AJ, Lee A, Rothermel G, Untch RH, Zapf C. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology* 1996; **5**(2):99–118. https://doi.org/10.1145/227607.227610
25. Mresa ES, Bottaci L. Efficiency of mutation operators and selective mutation strategies: an empirical study. *Software Testing Verification and Reliability* 1999; **9**(4):205–232. https://doi.org/10.1002/(SICI)1099-1689(199912)9:4<205::AID-STVR186>3.0.CO;2-X
26. Mathur AP. Performance, effectiveness, and reliability issues in software testing," in *Proceedings of the 5th International Computer Software and Applications Conference (COMPSAC79), Tokyo, Japan*, pp. 604–605, 1991. https://doi.org/10.1109/CMPSAC.1991.170248
27. Gligoric M, Zhang L, Pereira C, Pokam G. Selective mutation testing for concurrent code. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ACM: Lugano, Switzerland, 2013; 224–234. https://doi.org/10.1145/2483760.2483773
28. Kurtz B, Ammann P, Offutt J, Delamaro ME, Kurtz M, Gökçe N. Analyzing the validity of selective mutation with dominator mutants. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016*. ACM: Seattle, WA, USA, 2016; 571–582. https://doi.org/10.1145/2950290.2950322
29. Delgado-Perez P, Medina-Bulo I, Nuñez M. Using evolutionary mutation testing to improve the quality of test suites, in *2017 IEEE Congress on Evolutionary Computation (CEC)*, pp. 596–603, 2017. https://doi.org/10.1109/CEC.2017.7969365
30. Gopinath R, Ahmed I, Alipour MA, Jensen C, Groce A. Mutation reduction strategies considered harmful. *IEEE Transactions on Reliability* 2017; **66**(3):854–874. https://doi.org/10.1109/TR.2017.2705662
31. Byoungju C, Mathur AP. High-performance mutation testing. *Journal of Systems and Software* 1993; **20**(2):135–152. https://doi.org/10.1016/0164-1212(93)90005-I
32. Mateo PR, Usaola MP. Parallel mutation testing. *Software Testing Verification and Reliability* 2013. https://doi.org/10.1002/stvr.1471
33. Ma YS, Offutt J, Kwon YR. MuJava: an automated class mutation system. *Software Testing Verification and Reliability* 2005; **15**(2):97–133. https://doi.org/10.1002/stvr.308
34. Kim SW, Ma YS, Kwon YR. Combining weak and strong mutation for a noninterpretive Java mutation system. *Software Testing Verification and Reliability* 2013; **23**(8):647–668. https://doi.org/10.1002/stvr.1480

© 2021 John Wiley & Sons, Ltd

35. Mateo PR, Usaola MP. Reducing mutation costs through uncovered mutants. *Software Testing Verification and Reliability* 2015; **25**(5–7):464–489. https://doi.org/10.1002/stvr.1534
36. Mateo PR, Usaola MP. Mutant execution cost reduction: through MUSIC (Mutant Schema Improved with Extra Code), in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pp. 664–672, 2012. https://doi.org/10.1109/ICST.2012.156
37. Papadakis M, Malevris N. Automatic mutation test case generation via dynamic symbolic execution, in *2010 IEEE 21st International Symposium on Software Reliability Engineering*, 2010. https://doi.org/10.1109/ISSRE.2010.38
38. Schuler D, Zeller A. Javalanche: efficient mutation testing for Java, in *7th Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2009, pp. 297–298. https://doi.org/10.1145/1595696.1595750
39. Mateo PR, Usaola MP. Bacterio: Java mutation testing tool: a framework to evaluate quality of tests cases, *IEEE International Conference on Software Maintenance and Evolution, ICSM*, pp. 646–649, 2012. https://doi.org/10.1109/ICSM.2012.6405344
40. Hariri F, Shi A, Fernando V, Mahmood S, Marinov D. Comparing mutation testing at the levels of source code and compiler intermediate representation, in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, **2019**, pp. 114–124. https://doi.org/10.1109/ICST.2019.00021
41. Polo Usaola M, Reales Mateo, Pérez Lamancha B. Reduction of test suites using mutation, in *International Conference on Fundamental Approaches to Software Engineering*, Lecture Notes in Computer Science (LNCS, volume 7212). 2012.
42. Grindal M, Offutt J, Andler SF. Combination testing strategies: a survey. *Software Testing Verification and Reliability* 2005; **15**(3):167–199. https://doi.org/10.1002/stvr.319
43. “JavaParser-Home.” https://javaparser.org [accessed May 11, 2020].
44. Lima JAP, Guizzo G, Vergilio SR, Silva APC, Filho HLJ, Ehrenfried HV. Evaluating different strategies for reduction of mutation testing costs, in *Proceedings of the 1st Brazilian Symposium on Systematic and Automated Software Testing - SAST*, 2016, pp. 4.1–4.10. https://doi.org/10.1145/2993288.2993292
45. Myers G. *The Art of Software Testing*, Second ed. John Wiley & Sons, Inc.: Hoboken, NJ, USA, 2004.
46. Fraser G, Gargantini A. Experiments on the test case length in specification based test case generation, in *2009 ICSE Workshop on Automation of Software Test*, 2009, pp. 18–26. https://doi.org/10.1109/IWAST.2009.5069037
47. Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A. *Experimentation in Software Engineering*. Springer: New York, USA, 2012.
48. Choi BJ, DeMillo RA, Krauser EW, Martin RJ, Mathur AP, Offutt AJ, Pan H, Spafford EH. The Mothra tool set (software testing), in *[1989] Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences. Volume II: Software Track*, 1989, vol. **2**, pp. 275–284. https://doi.org/10.1109/HICSS.1989.48002
49. Luo Y, Ben K, Mi L. Software metrics reduction for fault-proneness prediction of software modules, in *Network and Parallel Computing*, 2010, pp. 432–441.
50. Yu P, Systa T, Muller H. Predicting fault-proneness using OO metrics. An industrial case study, in *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering*, 2002;99–107. https://doi.org/10.1109/CSMR.2002.995794
51. Ma YS, Kim SW. Mutation testing cost reduction by clustering overlapped mutants. *Journal of Systems and Software* 2016; **115**:18–30. https://doi.org/10.1016/j.jss.2016.01.007
52. Yu M, Ma YS. Possibility of cost reduction by mutant clustering according to the clustering scope. *Software Testing Verification and Reliability* 2019; **29**(1–2). https://doi.org/10.1002/stvr.1692
53. Ji C, Chen Z, Xu B, Zhao Z. A novel method of mutation clustering based on domain analysis, in *Proceedings of the 21st International Conference on Software Engineering & Knowledge Engineering (SEKE'2009), Boston, Massachusetts, USA*, *July 1-3, 2009*, 2009;422–425.

© 2021 John Wiley & Sons, Ltd