Automatic Testing of Design Faults in MapReduce Applications

Jesús Morán, Antonia Bertolino, Claudio de la Riva, and Javier Tuya, *Member, IEEE*

***Abstract*—New processing models are being adopted in Big Data Engineering to overcome the limitations of traditional technology. Among them, MapReduce stands out by allowing for the processing of large volumes of data over a distributed infrastructure that can change during runtime. The developer only designs the functionality of the program and its execution is managed by a distributed system. As a consequence, a program can behave differently at each execution because it is automatically adapted to the resources available at each moment. Therefore, when the program has a design fault, this could be revealed in some executions and masked in others. However, during testing, these faults are usually masked because the test infrastructure is stable, and they are only revealed in production because the environment is more aggressive with infrastructure failures, among other reasons. This paper proposes new testing techniques aimed to detect these design faults by simulating different infrastructure configurations. The testing techniques generate a representative set of infrastructure configurations that as whole are more likely to reveal failures using Random testing, and Partition testing together with Combinatorial testing. The techniques are automated by using a test execution engine called MRTest that is able to detect these faults using only the test input data, regardless of the expected output. Our empirical evaluation shows that MRTest can automatically detect these design faults within a reasonable time.**

***Index Terms*—Big Data, Combinatorial testing, MapReduce, Metamorphic testing, Partition testing, Random testing, Software testing.**

I. Introduction

In recent years, the volume of data generated by companies has grown exponentially and several challenges appear when it comes to storing, transporting and analysing such information. To overcome these challenges, new technologies are being created under the *Big Data* paradigm [1]. Their rise allows large scale analysis of data, from social web interactions to industrial sensor data, that can improve social and business performance.

There are several obstacles and challenges that affect this paradigm, such as the lack of skills [2]–[4], poor data quality [5] and different technological issues [3], [6], [7]. According to Gartner, it is expected that 60% of *Big Data* projects will fail to go beyond piloting and will be abandoned during 2017 [8].

The *MapReduce* processing model [9] stands out among *Big Data* applications. It is a key technology very broadly used by organizations [10] and implemented in several mature frameworks [11], [12], such as Hadoop [13], Flink [14], [15] and Spark [16], [17], among others. Because it is so widely adopted, the quality of *MapReduce* programs is important, especially for those employed in critical sectors as such as health (DNA alignment [18]) and security (image processing in ballistics [19]). An analysis over several months at Yahoo! indicates that around 3% of *MapReduce* programs are not finished [20], whereas another broader study places this percentage between 1.38% and 33.11% [21]. A study of 507 programs in production reveals at least 5 different kinds of faults [22], and other works [23], [24] have identified and classified more such faults that are caused by the incorrect design of *MapReduce* programs. Therefore, in this paper we propose new testing techniques to address these functional faults that are caused by incorrect design.

These types of faults include, but are not limited to, race conditions, computations with unavailable data because the distributed system allocates them to another computer, or automatic optimizations that remove data that are relevant to calculating the output. These faults are difficult to detect because they depend not only on the data, but also on how these data are executed in the large distributed architecture (infrastructure configuration): parallel executions, re-executions of some part of the data and optimizations, among others. In general, these non-deterministic faults are easy to mask in development/testing environments and go on to fail in more aggressive environments such as the production environment, thereby generating incorrect outputs or causing the program to crash.

In order to detect these kinds of faults in the early stages of development, our previous work simulates, in a test environment, the execution of the test cases in a thorough range of infrastructure configurations that could occur in production

Manuscript submitted August, 1, 2017; revised October, 2, 2017; accepted January, 27, 2017. This work was supported in part by the Spanish Ministry of Science and Technology under the PERTEST project (TIN2013-46928-C3-1-R), the Spanish Ministry of Economy and Competitiveness under both TestEAMoS (TIN2016-76956-C3-1-R) and POLOLAS (TIN2016-76956-C3-2-R) projects, the Principality of Asturias (Spain) under both the GRUPIN14-007 project and Severo Ochoa pre-doctoral grants (BP16215), the Italian MIUR under GAUSS project (PRIN 2015, 2015KWREMX), and ERDF funds.

J. Morán, C. de la Riva, and J. Tuya are with the Computer Science Department, University of Oviedo, Asturias, 33203, Spain (e-mail: {moranjesus, claudio, tuya}@ uniovi.es).

Antonia Bertolino is with ISTI-CNR, Pisa, 56124, Italy (e-mail: antonia.bertolino@isti.cnr.it).

(all potential configurations) [25], [26]. This paper extends the previous approach by proposing a more efficient strategy that automatically tests the program only under the relevant and representative configurations. Whereas in the previous work the selection of the configurations grows exponentially according to the number of input data records, the approach proposed here exploits combinatorial techniques that maintain a good number of failures detected within an acceptable time. In this way, the execution time and resource utilization of the test cases is clearly improved, allowing test cases with no limitation in the number of input data records, contrary to the previous work that only allowed a very small number of records. More specifically, the main differences between this paper and our previous work are: (a) Combinatorial, Partition and Random testing test strategies that instead of analysing all configurations exhaustively, only analyse a representative subset of configurations; (b) a metamorphic testing [27] approach to guide both the generation of configurations and the checking of failures, (c) whereas the previous work only supported a small volume of test input data due to the number of configurations growing exponentially, the current paper overcomes these practical issues and supports more test input data in less time, and (d) the paper includes a comprehensive validation through empirical experiment using real world programs. The contributions of this paper are:

1) Testing techniques based on combinatorial testing, partition testing and random testing to select the infrastructure configurations to be employed during the execution of the test cases.
2) Detection of the *MapReduce* design faults through an automatic partial oracle without any knowledge about either the program specification or expected output.
3) Automation of the testing execution in a test engine called MRTest that extends MRUnit [28] (XUnit for *MapReduce* programs).
4) Experimentation with 16000 test cases executed against 8 real-world programs to analyse the effectiveness of MRTest in detecting failures and its efficiency in executing the test cases.

The remainder of the paper is organized as follows. Section II introduces the *MapReduce* processing model. Related work is then discussed in Section III. The testing techniques proposed and the MRTest automatization are defined in Section IV. The experiment is performed and discussed in Section V. Finally, the conclusions and future work are detailed in Section VI.

II. Background Of MapReduce

MapReduce programs process large datasets distributed over several computers using the "divide and conquer" principle. In its simplest form, the *MapReduce* developer needs to implement only two components: the Mapper that splits one problem into several subproblems (Divide), and the Reducer that solves these subproblems (Conquer). During the execution, several instances of Mapper analyse the dataset in parallel and send to each subproblem the data needed to be solved. After all Mappers are executed, several instances of the Reducer are executed in parallel to solve the subproblems. Internally, the data are codified as <key, value> pairs, where the key is an identifier of a subproblem, and the value contains all the information that is needed to solve the subproblem. The developer designs the business logic based on the <key, value> pairs emitted from Mappers to Reducers. Finally, the output is a series of <key, value> pairs obtained through the deployment and execution of Mappers and Reducers over a distributed infrastructure.

More generally, a *MapReduce* program can be designed with more components. For example, a Combiner can be implemented to improve the performance by reducing the data exchanged between Mappers and Reducers. The Combiner is executed right after the Mapper with the aim of removing the <key, value> pairs that are irrelevant to solving the subproblem. The *MapReduce* applications can also be designed with other components such as, for example, the Partitioner that determines which Reducer analyses which <key, value> pair, the Sorter that controls the order of <key, value> pairs, and the Grouper that aggregates the values of each key before they are passed to the Reducer.

Distributed systems such as Hadoop execute the *MapReduce* programs in a non-deterministic way based on runtime factors, such as the resources available, observed infrastructure failures and other dynamic optimizations. Nevertheless, the same program with the same input data when executed in different infrastructure configurations should obtain the correct output. However, this is not always the case: in a prior work [23] we identified and classified several design faults that are raised in some infrastructure configurations but masked in others. Despite the fact that some authors suggest that the parallel programming must be deterministic by default (unless the developer explicitly indicates non-determinism) [29], this is not the case with these distributed systems.

To illustrate *MapReduce* and its executions, let us suppose the computation of the average temperature per year given a large dataset containing several years with their observed temperatures. This program can be designed in different ways. We suppose that the developer makes the following decisions. The problem of average temperature per year is divided into several subproblems where each subproblem calculates the average temperature of one year only (Decision 1). Then each subproblem is composed of one year with all temperatures of this year (Decision 2), and it is solved with the temperature average (Decision 3). The program includes a Combiner to improve the performance (Decision 4). With the foregoing decisions, the Mappers receive a subset of temperature data and emit <year, temperature of this year> pairs. Then the distributed system aggregates all values per key, that is, each subproblem grouped with all the data that needs to be solved. Therefore, the Reducers receive subproblems like <year, [all temperatures of this year]> and calculate the average temperature. After the Mapper, the Combiner can be executed, aimed at removing the irrelevant temperatures, and emitting their average.

The distributed system can execute the previous program in different ways, based on the runtime infrastructure configuration. For example, Fig. 1 shows three different executions with the following input: year 1999 with 4°, 2° and

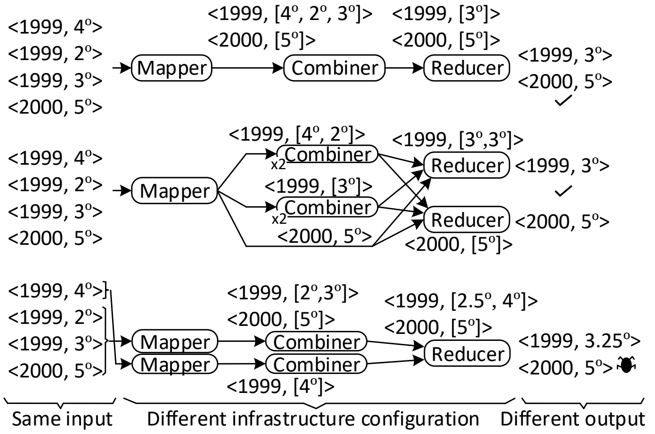


Fig. 1. Execution of MapReduce program that calculates the average temperature per year.

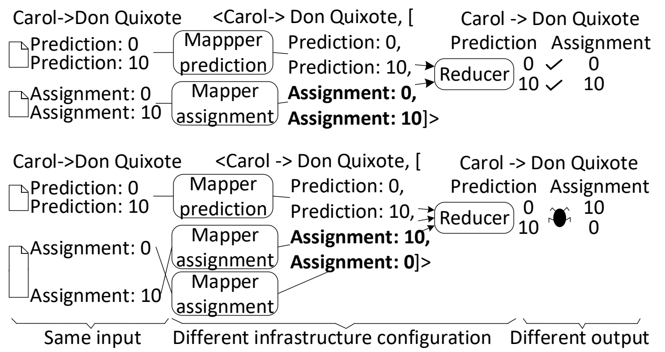


Fig. 2. Execution of a recommendation system in MapReduce.

3°; and year 2000 with 5°. Regardless of the infrastructure configuration, the program must obtain the right output: 3° as average in 1999, and 5° in 2000. The first configuration is the simplest with one Mapper, one Combiner and one Reducer. The Mapper analyses all temperatures and encodes them as <year, temperature>. Then the temperatures are grouped per year and sent to the Combiner that pre-calculates the average temperatures, and finally to the Reducer that obtains the correct output.

Depending on the runtime resources, the distributed system can execute the program automatically in more complex configurations. As we detail in Section IV, the configurations can have, among other things, a different number of Mappers and other automatic optimizations. For example, the second configuration of Fig. 1 is more complex than the first, employing one Mapper, two Combiners and two Reducers, also obtaining the correct output. We assume that the first Combiner receives the temperatures 4° and 2° of the year 1999, and emits their average, 3°. The second Combiner receives year 1999 with 3° and emits it, whereas the year 2000 with 5° is passed directly from Mapper to Reducer. After the Mappers and Combiners are executed, one Reducer analyses the temperatures of the year 1999 and another Reducer the year 2000. Eventually, this configuration also obtains the correct output.

In contrast, the third configuration that executes two Mappers, one Combiner per Mapper, and one Reducer does not obtain the right output. This execution obtains 3.25° as the average of 1999 rather than 3° due to a design fault. The developer makes some incorrect design decisions, among them, the use of <year, temperature> pairs and the Combiner to optimize the program. Both decisions are incompatible in this program because the Combiner replaces the temperatures locally available in each computer with their average, and then the Reducer cannot calculate the global average using only the local averages.

Although this program has a simple business logic, several developers make the previous incorrect design decisions to obtain the average temperature per year, as in the programs [30], [31]. The developer can fix the program by removing the Combiner, but this solution is not optimised. A better program design codifies the data as <year, {sum of temperatures, number of temperatures}> and uses a Combiner to update both the sum and number of temperatures [32].

A sample of a more subtle design fault is in the recommendation system Open Ankus [33]. The users assign points to a series of books, and then the system tries to forecast the points assigned for new books. The design fault is triggered during the calculation of the error between the user assignment and the system prediction. Fig. 2 depicts the execution of the program with the points assigned by Carol to the book Don Quixote and correctly predicted by the system. The first configuration with one Mapper for the predictions and one Mapper for the assignments obtains the correct output (the system predicts the result correctly). In contrast, when several Mappers for assignment are executed in parallel, the output of this program could potentially be faulty, depending on both the order of execution and how the data is distributed in parallel. For example, the program can be executed as in the second configuration of Fig. 2 obtaining the incorrect result that the system prediction is wrong.

When the business logic tends to be more complex, as in machine learning programs, it can be difficult for the developer to make the right design decisions, and the program may be prone to side-effects. An incorrect design in the *MapReduce* program may cause a failure in one of the different ways in which the distributed systems can execute the program. These design faults are difficult to detect during testing because they may depend on dynamic execution contexts. Thus they can be missed in the laboratory, but are then triggered in aggressive environments, such as a production environment with a mix of large data and infrastructure failures. When these aggressive situations happen, the distributed systems manage the execution with different mechanisms, such as re-executing part of the program or performing some optimizations that can reveal design faults. To avoid incorrect outputs in production, it is desirable to detect these program faults in the early stages of the development process.

III. Related Work

Software testing is among the most commonly used software quality-assurance techniques [34]. In recent years, this field has seen great progress [35], but concerning the testing of *Big Data* applications, there remain several challenges [36], [37]. In this domain, most works focus on performance testing [38], [39], but, as our previous example showed, functional testing is also important to avoid incorrect outputs [40]. In this paper we address functional testing.

As seen in the earlier examples, some faults depend on how distributed systems execute the programs according to the infrastructure configurations. If the program generates incorrect outputs in some configurations and the expected output in others, then the program has a design fault. A study of 507 *MapReduce* programs in production reveals at least 5 different kinds of design faults [22]. To detect them, Csallner et al. [41] and Chen et al. [42] use testing techniques based on symbolic execution and model checking. Other authors [23], [24] identified and classified other design faults that depend upon the infrastructure configurations. In our previous work, we proposed a test approach to detect such faults in the test environment by using a simulation of the infrastructure configurations [26]. In this paper we enhance on that work with a more practical technique that improves efficiency while still detecting the majority of faults by exploiting combinatorial strategies.

The production environment is composed of a large distributed infrastructure that over time exposes several failures [43]. In order to test in the same conditions as production, several research lines propose to inject infrastructure failures [44], [45] during testing, and several tools have been implemented to support their injection [46]–[48]. For example, Marynowski et al. [49] propose creating the test cases by specifying which computers fail and when. While some of the design faults can be detected by injecting infrastructure failures, others require a fine-grained control of the distributed system and the underlying large infrastructure. In the production environment it is difficult to control the execution of the test cases because at the same time other programs consume the resources of the computers and other infrastructure failures can happen that are beyond the tester's control. This paper does not inject failures, but simulates the different infrastructure configurations in a test environment, thereby obtaining fine-grained control and reproducibility of the tests.

Several research lines propose generating the test input data through different approaches: using data flow [50], based on a bacteriological algorithm [51], or with input domain analysis together with combinatorial testing [52]. Unlike these testing techniques, this paper does not focus on the generation of the test input data, but on simulating their execution in the infrastructure configurations that are more likely to reveal the faults. As such, this work is orthogonal to the above. The tester can use the previous approaches to obtain the test input data and then execute the test cases with the techniques proposed in this paper. As we have shown, the same program and the same input data executed in different configurations might produce different results so, apart from deriving a good test suite, the testing of *MapReduce* applications also requires the derivation of the correct configuration.

Several tools have been proposed to design and execute test cases for *MapReduce* applications. Herriot [53] allows the execution of the tests in a distributed infrastructure and at the same time supports the injection of infrastructure failures. Another tool called MiniClusters [54] executes the test cases in a distributed environment simulated in memory. For unit testing, MRUnit [28] provides an adaptation of JUnit [55] to the *MapReduce* processing model. All the above test tools only execute the test case in one infrastructure configuration and usually without parallelization. In this paper we devise a testing technique to generate and execute a representative set of infrastructure configurations that could occur in production and as a whole is more likely to reveal design faults. It is automated by means of an MRUnit extension, as described below.

IV. MRTEST: AUTOMATIC MAPREDUCE TESTING TECHNIQUE

In this section, we describe the test execution engine we propose called MRTest. Given a test input data, MRTest automatically generates the configurations aimed at revealing design faults (Section IV.A), then executes the test case in these configurations (Section IV.2), and finally checks if the program is faulty or not (Section IV.3).

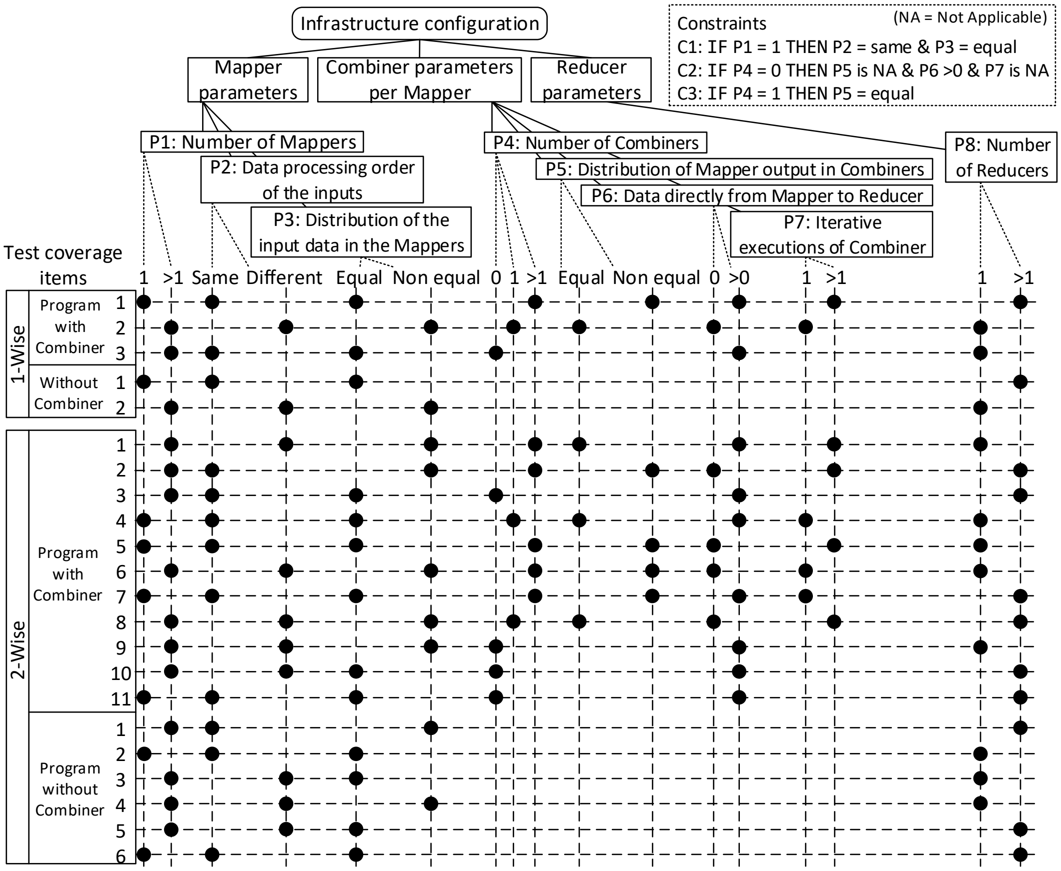
A. Generation of Infrastructure Configurations for Testing

In a prior work [26], we proposed an automatic technique that, given test input data, generates a thorough number of configurations (MRTest-Thorough). Then the faults are revealed when a failure occurs in one of these configurations. The previous technique has some limitations because it takes a long time and only supports a small volume of test input data. This paper proposes two new techniques that overcome these problems by reducing the number of configurations generated while maintaining the fault detection effectiveness: the two techniques use Random Testing [56] (MRTest-Random) and Partition Testing (Equivalence Partitioning [57] with Combinatorial Testing [58], [59]) (MRTest-t-Wise).

The first technique, **MRTest-Random**, generates the configurations randomly from all valid configurations. The tester indicates the number of configurations wanted, and MRTest-Random generates them randomly.

The second technique, **MRTest-t-Wise**, divides the set of all valid configurations into several partitions with similar behaviour and applies a combinatorial strategy to generate the configurations under test. In software testing, depending on the failure probability [60], Random testing can be as effective as Partition testing [61] and can be a feasible option [62]. In other circumstances, Partition Testing can be more effective than Random Testing [63]. As we discuss in Section V, our experiments show that both techniques MRTest-Random and MRTest-t-Wise can be effective in revealing *MapReduce* faults, but MRTest-t-Wise is significantly better. The latter testing technique is schematically represented in Fig. 3 and described below in three parts: (1) Division of the set of all valid configurations, (2) Combination strategy, and (3) Generation of the configurations.

Division of the set of all valid configurations: The set of all valid configurations is divided based upon the following parameters and constraints that are also represented at the top of Fig. 3:


Infrastructure configuration
Mapper parameters
Combiner parameters per Mapper
Reducer parameters
P1: Number of Mappers
P2: Data processing order of the inputs
P3: Distribution of the input data in the Mappers
P4: Number of Combiners
P5: Distribution of Mapper output in Combiners
P6: Data directly from Mapper to Reducer
P7: Iterative executions of Combiner
P8: Number of Reducers
Test coverage items
1-Wise
2-Wise

Constraints (NA = Not Applicable)
C1: IF P1 = 1 THEN P2 = same & P3 = equal
C2: IF P4 = 0 THEN P5 is NA & P6 >0 & P7 is NA
C3: IF P4 = 1 THEN P5 = equal

		Test coverage items	P1		P2		P3		P4			P5		P6		P7		P8	
			1	>1	Same	Different	Equal	Non equal	0	1	>1	Equal	Non equal	0	>0	1	>1	1	>1
1-Wise	Program with Combiner	1	●		●		●				●		●		●		●		●
		2		●		●		●		●		●		●		●		●	
		3		●	●		●		●						●			●	
	Without Combiner	1	●		●		●												●
		2		●		●		●										●	
2-Wise	Program with Combiner	1		●		●		●			●	●			●		●	●	
		2		●	●			●			●		●	●			●		●
		3		●	●		●		●						●				●
		4	●		●		●			●		●			●	●		●	
		5	●		●		●				●		●	●			●	●	
		6		●		●		●			●		●	●		●		●	
		7	●		●		●				●		●		●	●			●
		8		●		●		●		●		●		●			●		●
		9		●		●		●	●						●			●	
		10		●		●	●		●						●				●
		11	●		●		●		●						●				●
	Program without Combiner	1		●	●			●											●
		2	●		●		●											●	
		3		●		●	●											●	
		4		●		●		●										●	
		5		●		●	●												●
		6	●		●		●												●

Fig. 3. MRTest-t-Wise testing technique.

Mapper:

P1) Number of Mappers: 1 or >1. The program in production can be executed with one Mapper (1) that analyses the entire dataset, or alternatively with several Mappers that analyse different parts of the dataset in parallel (>1).

P2) Data processing order of the inputs: data are processed in the same order as they are encountered in the input (same), or in a different order (different). The *MapReduce* processing model does not guarantee that the data will be processed in the same order as they are stored in the input.

P3) Distribution of the input data in the Mappers: data equally distributed in the Mappers (equal) or not equally distributed (non equal). The Mappers process different subsets of input data: there could be configurations with an equal number of data in each Mapper, or with a different number of data.

Combiner:

P4) Number of Combiners per Mapper: 0, 1 or >1. Each Mapper can execute one Combiner (1), several Combiners (>1) or can emit the data directly to Reducer (0).

P5) Distribution of Mapper output in Combiners: data equally distributed in combiners (equal) or not equally distributed (non equal).

P6) Data directly from Mapper to Reducer: 0 or >0. All data emitted by Mappers can be pre-processed by the Combiner functionality (0), or in contrast some data can pass directly from Mapper to Reducer without executing the Combiner functionality (>0).

P7) Iterative executions of Combiner: 1 or >1. The output of the Combiner can be executed iteratively by the Combiner several times (>1) or only once (1).

Reducer:

P8) Number of Reducers: 1 or >1. The program can be executed in production with one Reducer that solves all subproblems (1) or with several Reducers that solve the subproblems in parallel (>1).

For example, the configuration at the bottom of Fig. 1 is characterized by the following parameters:

- P1 is >1: There are two Mappers.
- P2 indicates a different order: The input data is executed in a different order than they are stored in the input. The 4° temperature is executed after 2°, but in the input the temperature 4° is before 2°.
- P3 indicates a non-equal distribution of the data in Mappers: Each Mapper has a different number of input data. One Mapper has 1 register and the second has 3 registers.
- P4 is 1: Each Mapper only executes one Combiner.
- P5 indicates an equal distribution of the Mapper output in its Combiners. Each Mapper only has one Combiner that receives all its data, then the output of the Mapper is equally distributed in its Combiner.
- P6 is 0: There are no data that pass directly from the Mapper to the Reducer without the Combiner.

- P7 is 1: The Combiners are not executed iteratively several times, they are executed only once.
- P8 is 1: There is only one Reducer.

The configurations under test are obtained by a combination of the previous parameters. However not all combinations make sense, and to prevent non-meaningful combinations, we have derived the following constraints that descend from the *MapReduce* processing model:

- When the number of Mappers (P1) is 1, then: (a) Data processing order of the inputs (P2) is the same order as they are in the input, and (b) Distribution of the input data in the Mappers (P3) is equally distributed.
- When the number of Combiners (P4) is 0, then: (a) the distribution of the data in the Combiners (P5) is not applicable, (b) the Data directly from the Mapper to the Reducer (P6) is >0, and (c) the iterative executions of the Combiner (P7) is not applicable.
- When the number of Combiners (P4) is 1, then the data in the Combiners (P5) are equally distributed.

Combination strategy: Deriving all possible combinations of previous parameters is expensive and the t-Wise strategy (also known as t-Way) is applied [58], [64]. Instead of combining all the values of all parameters, t-Wise [65] combines only the values of all subsets of t parameters. For example, 1-Wise (each use) [66] requires that all values of each parameter appear in at least one test case, whereas 2-Wise (pairwise) requires that the combination of all values per pair of parameters appears in at least one test case. 2-Wise has been shown to be almost as good as all combinations of parameters [67] at detecting failures, but employing much fewer resources in terms of time and cost [68].

The MRTest-t-Wise technique generates the configurations covering the t-Wise combinations of the previous parameters and constraints. Fig. 3 details the configurations that must be covered (test coverage items) for 1-Wise and 2-Wise strategies. Each row of the figure represents a test coverage item that should be covered with a configuration that satisfies the parameters indicated by dots. The 1-Wise technique requires the generation of 3 configurations (test coverage items) when the program implements a Combiner, and 2 configurations in the other case. The 2-Wise is a more thorough combination, requiring 11 configurations for programs with a Combiner, and 6 when the program does not implement a Combiner.

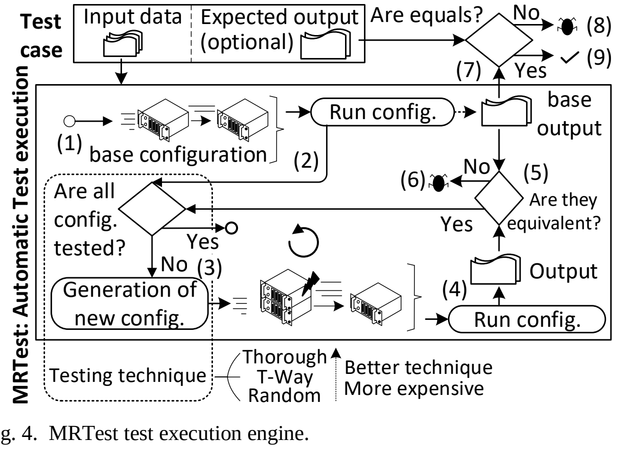


Fig. 4. MRTest test execution engine.

Generation of configurations: The configurations can be created manually to cover each test coverage item of the t-Wise selected, but the MRTest-t-Wise technique generates these configurations automatically. The following pseudo-code describes how the configurations are generated:

```
Input: t-Wise (testing technique selected, i.e. 2-Wise)
       sut (software under test)
Output:  Configurations that cover t-Wise in sut
(1)  Configurations ← ∅
(2)  tcis ← Get all test coverage items of the t-Wise
(3)  ∀ tci ∈ tcis
(4)  | Configuration ← ∅
(5)  | ∀ parameterToCover ∈ tci
(6)  | | value ← obtain randomly a value that covers
     | |                          parameterToCover in sut
(7)  | | IF value exists:
(8)  | |   Configuration ← Configuration ∪ value
(9)  | | ELSE  //When there is no value to cover
                                          parameterToCover
(10) | |   The actual configuration cannot cover the
     | |   test coverage item in sut, then backtracks
     | |   trying to generate again the configuration
     | |   with other values in previous parameters
     | |_  [maximum τ times (threshold)]
(11) |  IF Configuration covers the tci in sut:
(12) |_  Add Configuration to Configurations
(13) RETURN Configurations
```

In order to generate a configuration that covers each one of the test coverage items (1, 2, 3), the MRTest-t-Wise covers the first parameter with random values, then the second, and so on (4, 5, 6, 7, 8). For example, if the first parameter should be P1: >1, i.e., more Mappers, then a random value is selected greater or equal to 2 to guarantee >1 Mappers, and so on with the remainder of the parameters. Sometimes it can be impossible to cover one parameter because the test input data add semantic constraints unknown until the values selected in the previous parameters are executed (9, 10). For example, sometimes it is impossible to obtain a configuration with >1 Reducers because the test input data always lead to one Reducer. In these cases MRTest-t-Wise uses a backtracking approach. First it fulfils randomly the first parameter (4, 5, 6), then it executes the part of the program affected by this parameter (7, 8) and tries to cover the second, and so on. When the generator discovers at runtime that one parameter cannot be covered, then it backtracks, changing the value of previous parameters (9, 10). For example, a configuration can be created with 2 Mappers (P1 >1) but three input data cannot be equally distributed in them (it is impossible to cover P3 with equal distribution), then the generator backtracks changing the configuration to three Mappers (it maintains P1 >1 but changes randomly its value), and finally the three data items can be distributed equally in the Mappers. A threshold is set to prevent the generator from performing indefinitely or from backtracking for too long. When the threshold is overcome, then the technique does not create the configuration and the test coverage item is not covered (11, 3). By default, the threshold is 15 backtracks because we observed that usually when this number is exceeded then it is infeasible to cover the test coverage item, regardless of the set of valid configurations. Finally, those configurations that cover the test coverage items are generated (11, 12, 13).

B. Execution of Test Cases

In order to detect design faults, MRTest executes each test case in different configurations using one of the techniques described in the previous section (MRTest-Random and MRTest-t-Wise) or in the previous work (MRTest-Thorough) [26]. Then MRTest checks systematically that all configurations lead to equivalent outputs.

Given a test case with input data and, optionally, the expected output, the MRTest test execution engine is described in Fig. 4. First, MRTest executes the test input data in the base configuration (1), that is the simplest configuration with one Mapper, one Combiner and one Reducer without parallelization. Next, new configurations are iteratively generated (2,3) and executed (4) for a given testing technique selected by the tester: MRTest-Thorough, MRTest-t-Wise or MRTest-Random. The output obtained executing each configuration is checked against the output of the base configuration (5), revealing a fault if these outputs are not equivalent (6). Then MRTest can reveal faults with only the test input data, but the tester can optionally declare the expected output. In this case, the output of the base configuration is also checked against the expected output (7), detecting a fault when both are not equivalent (8, 9).

For example, let us revisit the program in Section 2 that calculates the average temperature per year. Fig. 1 describes the 1-Wise execution of a test case with the following test input data: year 1999 with 4°, 2° and 3°; and year 2000 with 5°. Firstly, MRTest generates and executes the base configuration (top of the figure) obtaining 3° as average in 1999, and 5° in 2000 (1, 2). Then MRTest generates and executes a configuration to cover the first test coverage item of 1-Wise (middle of figure), and again obtains the same output (3, 4, 5). In contrast, when it generates and executes the configuration of the third test coverage item (bottom of the figure), it obtains 3.25° as average of 1999 instead of the 3° obtained in the base configuration (3, 4, 5). Then MRTest automatically reveals a fault because the two outputs are different (6). We discuss further the oracle used in MRTest in the following section.

The test execution engine MRTest was implemented based on MRUnit library [28] maintaining its API and including new functions to indicate the testing technique to be used. This library is used to execute each configuration. In MRUnit, the test cases are executed with the base configuration, but this library is extended to generate other configurations and enable parallelism supporting the execution of several Mapper, Combiner and Reducer tasks. This test execution engine employs randomness to generate the configurations, but also supports pseudorandom numbers, also called seeds, to guarantee that the execution of the test case is reproducible in a deterministic way.

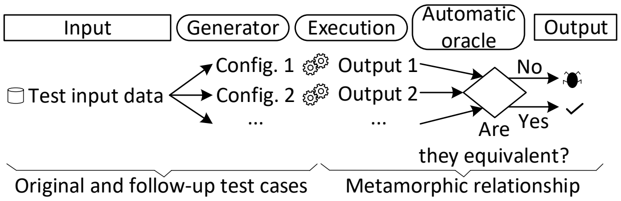


Fig. 5. Metamorphic oracle of MRTest.

C. Test Oracle

In software testing, the mechanisms that determine if the test reveals a fault or not are called test oracles [69]. There are some properties that characterize the efficacy of the test oracles [70], [71]. As discussed, if a design fault is present, the same program executed under different configurations can lead to different outputs. Based on this observation, the MRTest execution engine can reveal faults automatically even without knowing the expected output. It employs an automatic partial-oracle [69] that is derived from the program executions [72] using metamorphic testing [27]. Given a test case (original test case), metamorphic testing generates new test cases varying the original test case (follow-up test cases) to detect faults in a relationship amongst them (metamorphic relationship).

According to the software testing standard [73], a test case not only uses the test input, but also other test data that specify requirements for the test, such as databases, or configuration in the case of *MapReduce* programs. MRTest intends to detect those design faults that not only depend on specific test input, but also on specific configurations. For these faults, the test case must be designed with both the test input and the configuration in mind. MRTest receives the test input and then the metamorphic testing is focused in the relationships between the potential configurations. Given the test input, MRTest executes these test input data on the base configuration (original test case). Then MRTest generates the follow-up test cases maintaining the original test input but, but providing each one with different configurations. Finally, MRTest checks that both original and follow-up test cases lead to an equivalent output (metamorphic relationship). Whereas the metamorphic testing techniques usually generate the follow-up test cases by varying the test input, our approach generates the follow-up test cases by maintaining the test input but varying the configuration of the system under test.

MRTest can also be employed when the expected output is previously unknown or costly to obtain, as occurs in several machine learning programs [74]. Fig. 5 describes how the MRTest oracle can detect faults when given only the input data. The original test case is the test input data executed in the base configuration (1 mapper, 1 combiner, 1 reducer). Then MRTest generates and executes several configurations using the testing techniques described in the previous sections (follow-up test cases). Finally, it checks if their outputs are equivalent (metamorphic relationship), and if they are not then a potential fault is automatically detected.

According to the study of Segura et al. [75] the number of metamorphic papers will increase in years to come, but to date 49% employ the metamorphic testing capabilities in different problem domains, and only 2% present a tool. In our case, this paper not only defines and automatizes the metamorphic relationship to the *MapReduce* domain, but it also develops a tool that detects faults easily with only the test input data.

V. Experiments

The **goal** of these experiments is the evaluation of how, using different configurations in the execution of the test cases, the effectiveness in failure detection could be improved without significantly decreasing efficiency. The approach proposed in this paper, MRTest, executes the *MapReduce* test cases under several configurations, whereas the usual test execution engines, for example MRUnit, only execute them under a simple configuration. In the experiments, MRTest and MRUnit are compared in order to answer the following **research questions**:

RQ1) Do the test execution engines detect more failures when the *MapReduce* test cases are executed in different configurations?

RQ2) How expensive is the execution of the test cases in several different configurations?

The research questions are focused respectively on the effectiveness and efficiency during testing of the *MapReduce* programs. Depending on the field, the aptness of a technique can be referred to using different terms, for example: "effectiveness" for software testing techniques [76], "performance" for localization techniques [77], or "accuracy" for the classification techniques of machine learning [78]. In this paper, we use the term "effectiveness" regarding the quantity of failures detected, and "efficiency" regarding the execution time employed by the techniques. The planning and the results of the related experiments are presented in the next two subsections, and the discussion of the experiments together with the limitations in Subsection V.C.

A. Effectiveness Experiments

The **goal** of the effectiveness experiments is the assessment of how many failures are detected. Following the Basili et al. [79] template, the goal is: *Analyze* the test case execution engines (MRUnit and MRTest) *for the purpose* of evaluation *with respect to* their respective effectiveness in detecting failures due to a program design fault against the *MapReduce* processing model *from the point of view of* the tester and developer *in the context of Big Data* applications. The **planning** of the experiments is described in Subsection V.A.1 and their **results** are reported in Subsection V.A.2.

1) Effectiveness: Setup

In this experiment, 8000 different test cases from 4 real world programs are executed in MRUnit and MRTest in order to analyse their capability to detect failures. Each one of the 4 programs has a known design fault that is only revealed in some of the potential configurations and masked in others. The programs used, including a summary of the functionality and the cause of the faults, are:

1) Open Ankus [33]: A recommendation system that predicts for each user the items that could be of interest to them (films, books, cities, and so on), based on choices of other users and their similarities to the user in question. This program could fail when the data of each user-item is split and parallelized.
2) Data quality analysis [80]: Measure of the quality of data interchanged between companies, based on international standards. This program did not correctly track the measurements and they could be incorrectly assigned due the parallel execution. The production version of this program has removed the fault.
3) Movie analysis [81]: Statistics analysis of movies, based on the ratings of users. This program is implemented with an incorrect Combiner.
4) Data cleaner Knn analysis [82]: Knn machine learning algorithm to clean text data, based on the number of transformations, insertions and removals of incorrect letters in the words of the text. This program fails when one Mapper needs data that are not locally available because it is assigned to another Mapper.

For each program, 2000 test cases that contain data able to trigger the faults are executed in MRUnit and MRTest with different modes. The test cases are generated iteratively with random data until we have 2000 test cases that are able to trigger the fault. Because the program faults are known, all of the potential test cases are automatically analysed in order to check if the data can generate incorrect outputs under at least one configuration of the *MapReduce* configurations. For example, the program described in Section 2 that calculates the average temperature of each year, has a design fault that is not revealed by all inputs. We can automatically check if an input data is able to trigger the fault because the failure is only raised when the average of temperatures is different to the global average of local averages.

The **population** of the experiment is composed of all test cases with data able to trigger these faults in some configurations. Each of these test cases is then taken as the **experimentation unit**, and the **observation** is whether the test execution engines detect a failure or mask the fault. The **dependent variable** or response variable is the rate of failures detected by the different execution engines, which are the **independent variable**. The **baseline** is MRUnit and the **treatments** are MRTest executed in the following modes:

- 1-Wise: Based on the test coverage items proposed in MRTest-1-Wise algorithm, executes 3 or 2 configurations depending on whether the program has a Combiner or not, respectively.
- 2-Wise: Based on the test coverage items proposed in MRTest-2-Wise algorithm, executes 11 or 6 configurations depending on whether the program has a Combiner or not, respectively.
- 0-Random that executes randomly one configuration (MRTest-Random), in order to compare fairly with MRUnit that also executes one configuration (one Mapper, one Combiner and one Reducer).
- 1-Random in order compare fairly with 1-Wise. Executes 3 or 2 configurations depending if the program has a Combiner or not, respectively.
- 2-Random in order to compare fairly with 2-Wise. Executes 11 or 6 configurations depending if the program has a Combiner or not, respectively.

MRTest-thorough is not analysed in the experiments due to its limitations, such as only supporting a small amount of test input data or taking a long time to execute a test case. There are other elements that could affect the experiment and are treated as **blocking factors**:

- The size of the test input data could affect the rate of failures detected, so two sizes of data are considered: a small size (between 1 and 10 <key, value> pairs) and a larger size for functional testing purposes (between 11 and 35 <key, value> pairs).
- The generation of the configurations in MRTest is based on some pseudorandom functionality that could introduce noise in the failure rate. During the experiments, the different test engines employ the same pseudorandom number generated also in a pseudorandom way.

In the experiments two **sampling methods** are used: consecutive sampling to select the *MapReduce* programs and random sampling to select the test cases. Ideally the subject programs should be selected randomly, but as in the case in many software engineering experiments, this is not viable [83]. As such 4 real world programs that contain a known fault are selected instead.

As stated above, for each one of these programs, 2000 test cases that contain data able to trigger the fault are generated randomly. We grouped them in **trials** of 100 test suites with 20 test cases each: 50 test suites contain test cases with input data between 1 and 10 <key, value> pairs, and the other 50 test suites between 11 and 35 <key, value> pairs due to a pre-established blocking factor. All of these test suites are executed in the baseline (MRUnit) and the five treatments (MRTest), and then the rate of the failures detected is observed[1].

In these experiments, the **effectiveness** is measured by the percentage of failures detected per test suite. Then the effectiveness of the execution engines is compared via the **statistic test** Wilcoxon Sign Rank Test. This non-parametric statistic test analyses if there are significant differences based on the medians, then the null **hypothesis** is defined as H_0: $median(Effectiveness)_{MRUnit} = median(Effectiveness)_{MRTest}$

2) Effectiveness: Results and Discussion

Table I summarizes the number of test cases which detect a failure by each test execution engine (MRUnit and MRTest) during the experiments. This table shows that design faults against the *MapReduce* processing model are not detected in general by MRUnit, whereas MRTest approaches are able to detect them. The number of test executions that detect a failure by MRUnit is almost 0% whereas even considering the weakest MRTest approach, 0-Random, more than 15% of the test cases detect a failure; the strongest MRTest approach, 2-Wise, catches a failure in more than 60% of test cases. In general terms, 1-Wise and 1-Random detect more failures than MRUnit, and finally 2-Wise and 2-Random detect the majority of failures, regardless of the number of <key, value> pairs in the input data. In all approaches, the execution time of each test case is reasonable, being in the order of a few milliseconds/seconds. As we explain in detail in the following subsections, the majority of the test cases take less than 1 second to be executed in MRTest, regardless of the approach employed.

During the experiments, MRUnit only detects 4 faults out of 8000, having an effectiveness of 0 in Table I due to rounding to

TABLE I
EFFECTIVENESS OF FAILURE DETECTION OF 100 TEST SUITES OF 20 TEST CASES FOR EACH ONE OF THE 4 REAL WORLD PROGRAMS WITH FAULT

Program	Treatment	[1-10] <key, value> pairs		[11-35] <key, value> pairs		Total	
		Number of test cases that detect a failure	Effectiveness	Number of test cases that detect a failure	Effectiveness	Number of test cases that detect a failure	Effectiveness
Open Ankus	MRUnit baseline	0	0.00	0	0.00	0	0.00
	0-Random	307	0.30	293	0.30	600	0.30
	1-Wise	490	0.50	302	0.30	792	0.40
	1-Random	513	0.50	479	0.50	992	0.50
	2-Wise	754	0.75	620	0.60	1374	0.70
	2-Random	898	0.90	861	0.85	1759	0.90
Data quality analysis	MRUnit baseline	1	0.00	3	0.00	4	0.00
	0-Random	773	0.75	900	0.90	1673	0.85
	1-Wise	816	0.80	954	0.95	1770	0.90
	1-Random	925	0.95	984	1.00	1909	0.95
	2-Wise	994	1.00	1000	1.00	1994	1.00
	2-Random	992	1.00	1000	1.00	1992	1.00
Movie analysis	MRUnit baseline	0	0.00	0	0.00	0	0.00
	0-Random	169	0.15	183	0.20	352	0.15
	1-Wise	562	0.55	395	0.40	957	0.48
	1-Random	378	0.35	328	0.30	706	0.35
	2-Wise	952	0.95	861	0.85	1813	0.90
	2-Random	694	0.70	534	0.53	1228	0.63
Data cleaner Knn analysis	MRUnit baseline	0	0.00	0	0.00	0	0.00
	0-Random	876	0.75	946	0.95	1822	0.90
	1-Wise	983	0.80	1000	1.00	1983	1.00
	1-Random	978	0.95	997	1.00	1975	1.00
	2-Wise	1000	1.00	1000	1.00	2000	1.00
	2-Random	1000	1.00	1000	1.00	2000	1.00

Effectiveness = Median of percentage of failures detected per test suite (measured between 0 and 1).

[1] This type of experiment design is called “within subject design with post-test”

two decimal places. These faults are detected because MRUnit sorts the <key, value> pairs when the base configuration is executed and sometimes this change is enough to detect the faults. The execution of the test cases under different configurations can reveal design faults whereas the execution under one configuration could mask them, as occurs in MRUnit. Fig. 6 shows for each program the differences between the tests execution engines in terms of the effectiveness (percentage of failures detected per test suite). This figure uses a violin plot that shows the probability density function and gives a reference with a boxplot. The best testing techniques at detecting failures are 2-Wise and 2-Random, followed by 1-Random and 1-Wise, then 0-Random, and finally MRUnit, which hardly detects any design failures. According to the Wilcoxon Sign Rank Test, all MRtest approaches are significantly better at detecting failures than MRUnit.

In order to compare the best approaches, the Wilcoxon Sign Rank Test is also applied in each program between 2-Wise and 2-Random. Considering the Data Quality Analysis and Data Cleaner Knn Analysis programs, there is no significant difference between 2-Wise and 2-Random (p-value$_{[1-10]}$=0.69, p-value$_{[11-35]}$=1, p-value$_{[1-10]}$=1 and p-value$_{[11-35]}$=1, respectively), for the Open Ankus program 2-Random is better (p-value$_{[1-10]}$=2.7e-09 and p-value$_{[11-35]}$=2.8e-09) and for the Movies Analysis program 2-Wise is better (p-value$_{[1-10]}$=3.7e-10 and p-value$_{[11-35]}$=3.8e-10).

Fig. 7 shows the aggregation of the data for the 4 programs, 2-Wise being the best approach in detecting failures with a significant difference compared with 2-Random (p-value=0.0043).

In terms of failure detection effectiveness, all MRTest approaches are better than MRUnit, with 2-Wise and 2-Random standing out, followed by 1-Random and 1-Wise, and finally 0-Random.

B. *Efficiency Experiments*

The **goal** of the efficiency experiment is the assessment of how much time is spent during the execution of the test cases. Following the Basili et al. [79] template the goal is: *Analyze* the test case execution engines (MRUnit and MRTest) *for the purpose* of evaluation *with respect to* their efficiency in executing the test cases of the *MapReduce* programs *from the point of view of* the tester and developer *in the context of Big Data* applications. The **planning** of the experiments is described in Subsection V.B.1 and their **results** in Subsection V.B.2.

1) Efficiency: Setup

In this experiment, 16000 different test cases from 8 real world programs are executed in MRUnit and MRTest in order to analyse the execution time expense per test case. Half of these programs have design faults and their test cases are re-used from the previous experiment, the other 4 programs have no known faults and their functionality is summarized below:

5) Graph clustering [84]: Algorithm to cluster the connected nodes in graphs.
6) Phonetic analysis [82]: Algorithm to clean text data based on the similarities and differences between the phonetic pronunciation.
7) Goldstein analysis [85]: Measure of the conflicts and cooperation between countries based on the Goldstein

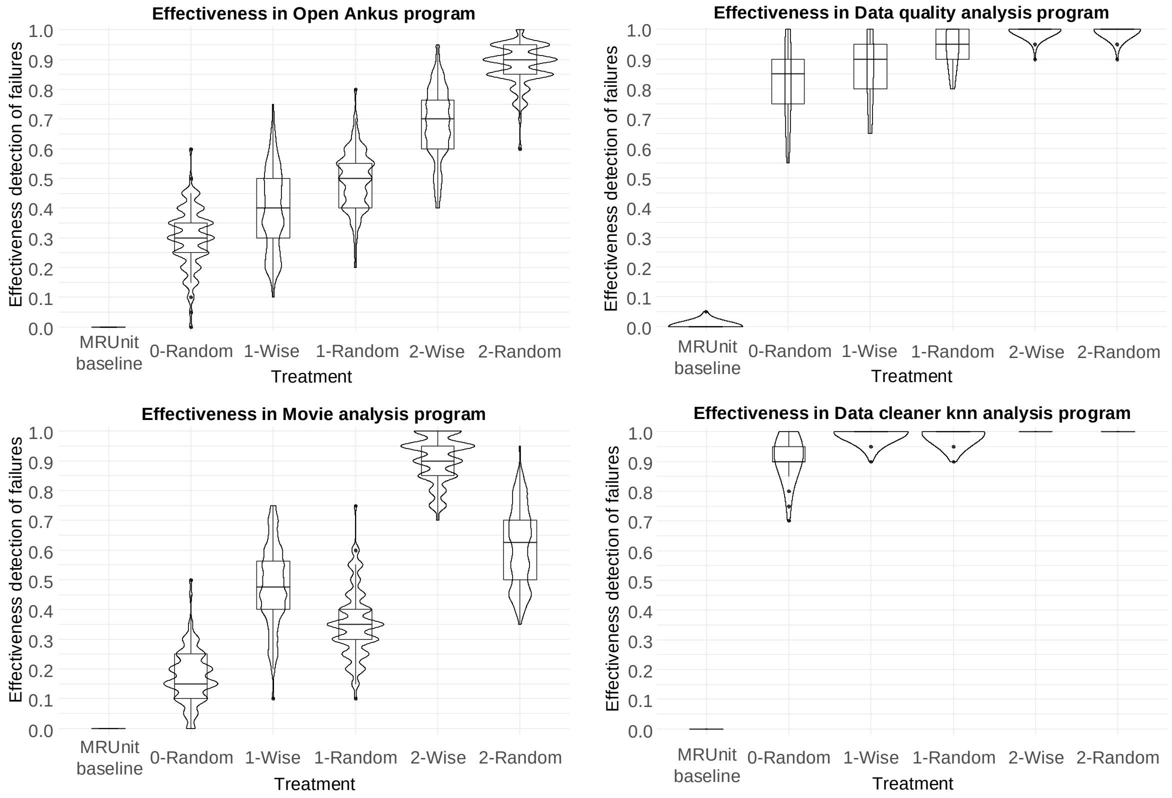


Fig. 6. Distribution of percentage of failures detected per test suite in each program.

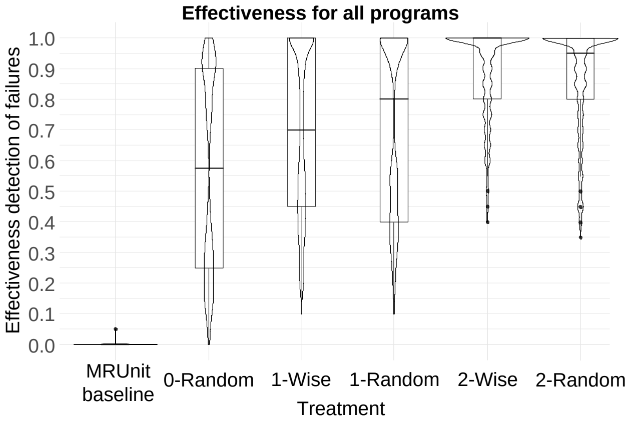


Fig. 7. Distribution of percentage of failures detected per test suite (effectiveness) in all programs.

code.

8) Restaurant analysis [86]: Finds restaurants by cuisine located in safe/unsafe zones in New York.

For each of these programs, 2000 test cases are generated randomly and executed in MRUnit and MRTest with different modes. The **population** is composed by all possible test cases of the *MapReduce* programs, and each one is the **experimentation unit**. All test cases are executed in order to analyse the execution time (**observation**). As in the previous experiment, sets of 20 test cases constitute the test suites that are executed in 6 test engines.

The **dependent variable** or response variable is the execution time of the test case by the different execution engines (**independent variable**). The **baseline** is MRUnit and the **treatments** are MRTest executed in the same way as the previous experiment: {0,1,2}-Random, {1,2}-Wise.

In this experiment, there are other variables that could affect the results and they are treated as **blocking factors** (note that the first two factors were also considered in the previous experiments):

- The size of the input data affects the execution time. The following number of <key, value> pairs are considered during the experiments: between 1 and 10, and between 11 and 35.
- The pseudorandom functionality of the MRTest could introduce noise. To avoid it, the same pseudorandom numbers are used in the MRTest and are generated in a pseudorandom way.
- MRTest executes the test cases with different configurations until a failure is detected or the maximum number of configurations of the approach is reached. Therefore, the execution time can be different whether the program has a fault or not. In this experiment two types of programs are considered: 4 programs with faults reused from the previous experiment and 4 programs without known faults described in this section.
- The execution time could depend on the resources of the computer. All test cases are executed in a commodity computer with a CPU Intel Core i5, 3.20GHz Windows 10 x64, and Java 1.8 with memory generated dynamically up to 250MB.

In order to detail the differences in the execution time, this experiment analyses **descriptive statistics**: a regression model of the execution time in terms of the number of input <key, value> pairs.

As in the previous experiment, the **sampling methods** are consecutive sampling of 8 real world programs and random sampling for the test cases. The number of **trials** per program is again 100 test suites of 20 test cases divided in two sizes of input data: from 1 to 10 <key, value> pairs, and from 11 to 35 <key, value> pairs. Each of these test suites is executed in the MRUnit (**baseline**) and MRTest with different parameters (**treatments**)[2].

2) Efficiency: Results and Discussion

MRUnit is the most efficient approach because it only executes one Mapper, one Combiner and one Reducer, whereas MRTest executes several of these configurations in order to reveal more faults simulating a production environment. Table II summarizes the average execution time of test cases in programs with and without known design faults. MRTest executed in 2-Wise mode is a better approach for detecting

TABLE II
AVERAGE EXECUTION TIME OF TEST CASE THROUGH 100 TEST SUITES OF 20 TEST CASES FOR EACH ONE OF THE 8 REAL WORLD PROGRAMS, IN MILLISECONDS

Input size	Treatment	Programs with known faults				Programs without known faults			
		Open Ankus	Data quality analysis	Movie Analysis	Data cleaner Knn analysis	Graph clustering	Phonetic analysis	Goldstein analysis	Restaurant analysis
[1-10] <key, value> pairs	MRUnit baseline	51.0	50.9	53.0	54.4	4.9	3.7	3.3	3.8
	0-Random	69.0	69.2	75.1	64.4	193.4	44.1	8.7	7.5
	1-Wise	84.2	94.3	1780.7	80.2	769.9	131.6	21.4	33.3
	1-Random	72.2	72.4	89.2	65.0	501.7	73.2	13.3	11.5
	2-Wise	149.9	145.0	2248.9	70.6	5140.7	384.3	74.8	563.1
	2-Random	77.4	73.5	140.0	64.7	1855.1	195.0	33.9	27.1
[11-35] <key, value> pairs	MRUnit baseline	51.8	50.6	55.3	78.8	7.7	4.8	5.4	5.3
	0-Random	76.0	75.3	93.2	115.0	1183.7	283.7	19.4	16.6
	1-Wise	85.5	104.8	139.4	152.2	3141.6	431.6	29.8	49.9
	1-Random	84.0	78.9	157.7	117.4	3445.0	539.6	36.5	28.6
	2-Wise	128.6	117.3	468.3	123.5	15013.0	1357.9	118.2	2282.4
	2-Random	104.0	78.8	575.7	117.2	13161.2	1606.6	153.5	105.2

[2] This type of experiment design is called "within subject design with post-test"

failures than Random, but it usually takes longer. In the test cases executed during the experiments, MRUnit takes, on average, a few milliseconds to execute a test case, whereas MRTest usually takes a few milliseconds-seconds, depending on the program and the data that are received. When the program has a fault and MRTest detects it, the execution time is quite similar to MRUnit (x2 or x3) because MRTest finishes after the execution of few configurations. In the case that MRTest does not detect a fault, the execution time on average increases by x200 or x400 from MRUnit, but it remains in the order of milliseconds-seconds per test case.

Given a program, there are several test cases that take more time than others, especially when {1,2}-Wise does not cover the test coverage items after trying to generate several configurations. The most expensive test case takes 4.5 minutes for the previous reasons, but in general the test cases are executed in milliseconds or a few seconds, depending on the program functionality and the input received. As Fig. 8 depicts, 75% of the test cases are executed in less than 1 second and 90% in less than 4 seconds.

The execution time depends on several factors, but it increases according to the number of <key, value> pairs in the test case. In Fig. 9 the trend of the execution time based on the number of <key, value> pairs is described for the 4 programs with faults, and in Fig. 10 for the other 4 programs without known faults. This trend in general has more slope in 2-Wise, 1-Wise and 2-Random because these approaches generate and execute more configurations. In these approaches the execution time is more dispersed because it does not only depend on the number of <key, value> pairs, but also on the program and on the data processed. In the case of {1,2}-Wise, the execution time also depends on the non-covered test coverage items, because, for example, in the most expensive test cases, MRTest takes a long time trying to generate values that cover the configurations that cannot be covered. For this reason, the execution time of 2-Wise in the Open Ankus and Data Quality analysis programs decreases according to the input size. When these two programs receive a small amount of input data, the 2-Wise takes time trying to cover the test coverage items. In some cases, 1-Wise is more expensive than 2-Wise because the test coverage items are different. For example, in the Data cleaner

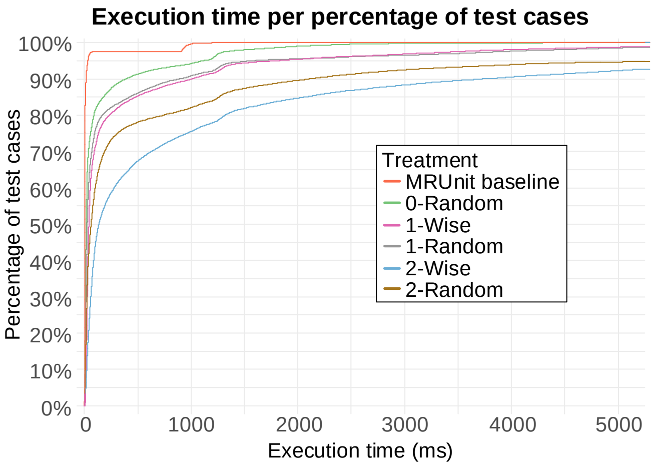


Fig. 8. Accumulated distribution of the test cases execution time.

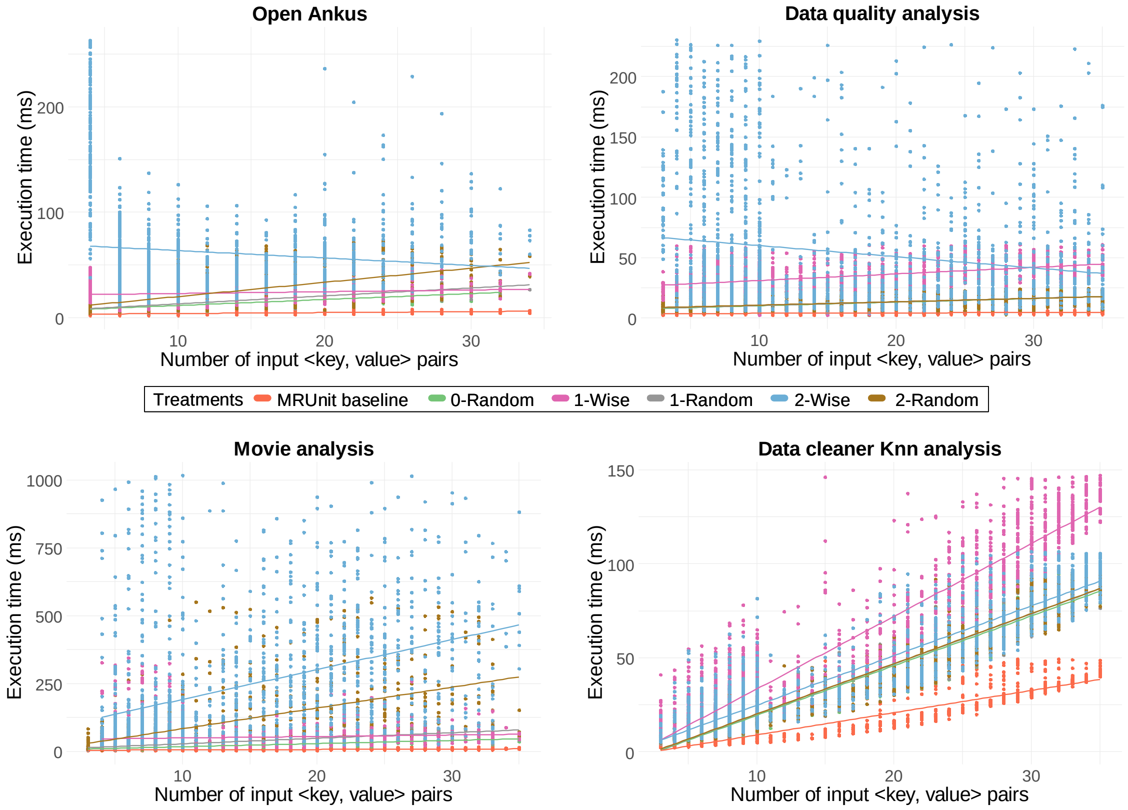


Fig. 9. Execution time of the test cases of programs with known faults according to the number of <key, value> pairs.

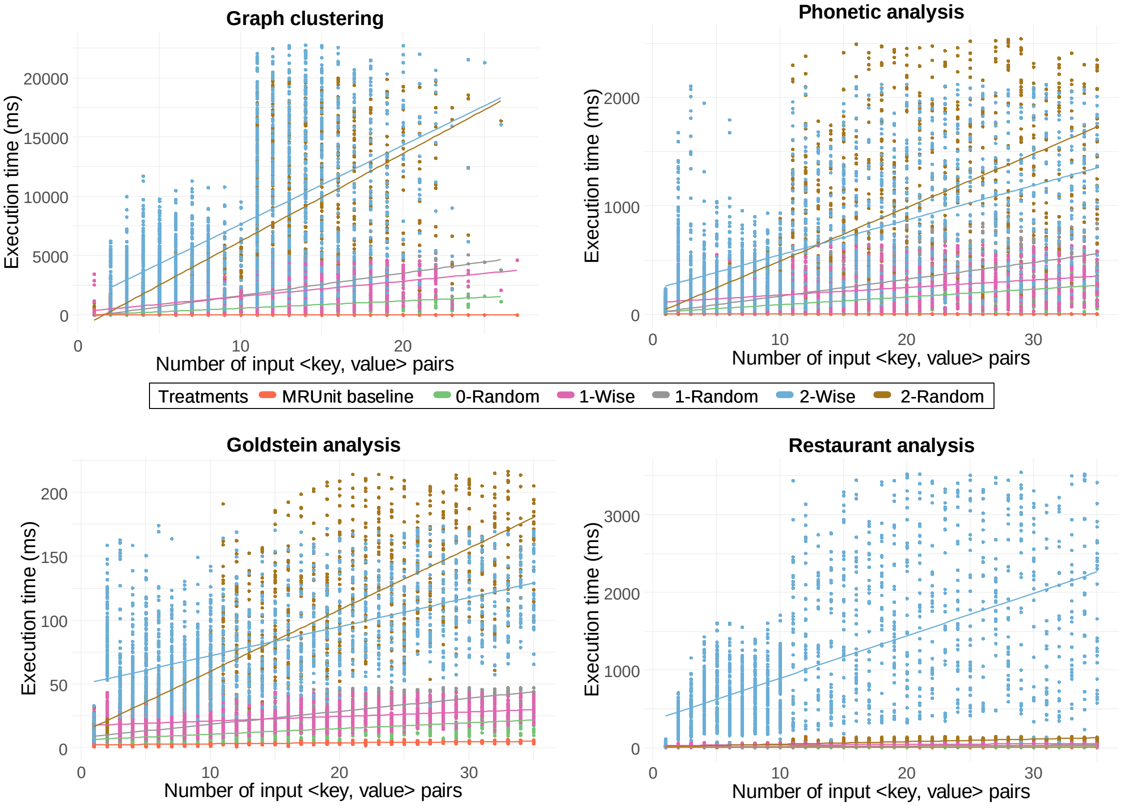


Fig. 10. Execution time of the test cases of programs without known faults according to the number of <key, value> pairs.

Knn analysis program, the second test coverage item of 1-Wise cannot be covered (it requires only one Reducer and the program guarantees several) but the approach wastes time trying to cover it.

While MRUnit is not intended to detect these design faults, all approaches of the MRTest are effective enough detecting them, particularly 2-Wise mode. These approaches take a few milliseconds-seconds to execute the test cases and could be a reasonable alternative to detect design failures before they are encountered in production.

C. *Discussion of Results*

The experiments indicate that both test execution engines proposed in this paper, MRTest-Random and MRTest-t-Wise, are able to detect within an acceptable time a broad number of failures that are caused by the non-deterministic executions of *MapReduce* programs. Of the two, the MRTest-2-Wise is significantly better at failure detection, and takes an acceptable amount of time to complete the tests as well. In contrast, the MRUnit test execution engine employs less time but it hardly detects any of these types of failures. The remainder of this subsection discusses the limitations of these experiments, including the internal, external and construct threats of validity and their subcategories [83], [87], [88].

The **internal** threats are those issues regarding the causal relationship between independent variables and dependent variables. One part of the experiments analyses the execution time, but some noise can be introduced into the measurements by other operative system tasks (*Confounding effects of variables*). To mitigate this problem, the experiments are executed in the same computer without any other programs operating in the background.

The tool that automates the research, MRTest, can contain faults and other limitations. To mitigate the potential faults of the tool, manual/automatic testing was performed mainly from the functional and performance point of view. This tool may cause side-effects in the programs that perform some communications with external services that are outside the testing context. For example, when the program under test inserts data in an external database, MRTest can perform the insertions for each of the configurations executed. When the external service is fully controllable, then the tester can handle the side-effects inside the test cases.

The **external** threats are those issues that can affect the generalization of the results. The subjects of this experiment are 16000 test cases randomly selected from 8 *MapReduce* programs selected by consecutive sampling. Ideally, the programs should also be selected randomly, but often this is not feasible in software engineering (*Interaction of selection and treatment*). For *Big Data* programs, there is no benchmark of faults and industrial programs are not usually available. This problem is mitigated by using some real-world applications, instead of using programs with seeded faults (hand-seeded faults or mutation faults) that are prone to other external threats [89], [90]. Therefore, there are other issues regarding seeded faults when they are used to evaluate testing techniques. The hand-seeded faults are injected by the expert and they are subjective, decrease the reproducibility of the experiments and are not representative of real faults in terms of easy detection [91]. In contrast, mutation faults are representative of the majority of faults, but this is not the case when the developer implements an incorrect algorithm [92]. The faults pursued by this paper fall into the previous category of faults that are not possible to substitute with mutations. The faults that are the target of this paper are caused by incorrect design decisions that lead to the implementation of faulty algorithms, completely different from those of the correct implementation. As such, the injection of mutation faults is not a feasible way to evaluate the testing techniques of this paper.

The tool that automates the research, MRTest, does not fully support the testing of non-deterministic programs (*Applicability of results across different samples*). This research proposes the execution of the test case in different configurations and finally a metamorphic relationship checks if their outputs are equivalent. The tool only checks if the outputs are equals or not, but this is not enough for non-deterministic programs. To avoid this problem, the tester can implement a function to check if two non-equal outputs are equivalent or not in the non-deterministic program. There are also metamorphic relationships for non-deterministic programs [93], [94].

Other results can be obtained if MRTest generates the configurations in a different way (*Applicability of results when technique is varied*). The configurations are generated based on the combination of different parameters, but there could be more parameters not considered or better ways to generate the configurations such as, for example, using a search-based approach.

The **construct** threats are those issues between the experiment and its underlying theoretical concepts. The test execution engines proposed are only compared against MRUnit despite the fact that there are other ways to automate the testing execution. In general, MRUnit is more standardized and controllable when performing tests in the *MapReduce* applications.

One part of the experiment analyses the efficiency of the test execution engine based only on the execution time measure, but there could be more measures not considered, such as memory (*Mono-operation bias*). To mitigate this problem, the experiments were executed in a commodity computer with few resources. The memory does not appear relevant because its usage was low during the experiments. Furthermore, the tool that automates the research was tested to avoid memory bottlenecks, and some memory leaks of MRUnit were removed.

VI. Conclusions and Future Work

The detection of design faults in *MapReduce* depends on the test input data and on the test configurations, i.e. how the test data are executed in parallel. These design faults can be revealed in some executions and masked in others. Thus, although the application may appear to work correctly in the test environment, this might not be the case when it is passed to production because usually these faults are only revealed in aggressive environments. In this paper, we presented two black-box testing techniques that automatically detect these faults. Given a set of test input data, the testing techniques simulate the execution in infrastructure configurations aimed at revealing the faults, and check that all executions lead to equivalent outputs. These testing techniques are automated in a test execution engine called MRTest.

We performed an empirical study to evaluate the effectiveness and efficiency of the testing techniques proposed (MRTest-Random and MRTest-t-Wise) compared to the XUnit tool of *MapReduce* programs (MRUnit). The results showed that our approaches are more effective in detecting faults while still employing reasonable time. The results also showed that MRTest-t-Wise based on Partition testing detects faults with a significantly lower fraction of tests than MRTest-Random that is based on Random testing.

MRTest enables fine-grained control of the test case execution at the same time as it guarantees its reproducibility in the same circumstances. The simulation of the test case in different production environments can be carried out in a non-intrusive way and with few resources, deploying MRTest on a commodity computer in the laboratory. Furthermore, the testing techniques of this paper are easy to use because they do not need the expected output to reveal the faults, only the test input data.

Big Data programs have large quantities of data that can be used as test input data. As part of our future work, we plan to complete test automation by taking advantage of the production data at runtime (online) or before runtime (offline). Another research line pursues the automatic diagnosis and localization of these faults. We also plan as future work to adapt the testing techniques of this paper beyond *MapReduce*, such as in the Lambda architecture, in *Big Data* streaming frameworks or in data-flow paradigms considering their similarities to *MapReduce*.

References

[1] ISO/IEC JTC 1 - Big Data, preliminary report. 2014.

[2] Xerox, “Big Data in Western Europe Today,” 2015.

[3] Capgemini Consulting, “Big Data survey,” 2014.

[4] B. Marr, “Where Big Data Projects Fail,” 2015. [Online]. Available: http://www.forbes.com/sites/bernardmarr/2015/03/17/where-big-data-projects-fail/. [Accessed: 31-Jan-2018].

[5] N. Laranjeiro, S. N. Soydemir, and J. Bernardino, “A Survey on Data Quality: Classifying Poor Data,” in *Proceedings - 2015 IEEE 21st Pacific Rim International Symposium on Dependable Computing, PRDC 2015*, 2016, pp. 179–188.

[6] V. Marx, “Biology: The big challenges of big data,” *Nature*, vol. 498, no. 7453, pp. 255–260, Jun. 2013.

[7] D. Bachlechner and T. Leimbach, “Big data challenges: Impact, potential responses and research needs,” in *2016 IEEE International Conference on Emerging Technologies and Innovative Business Practices for the Transformation of Societies (EmergiTech)*, 2016, pp. 257–264.

[8] Gartner, “How to Take a First Step to Advanced Analytics,” 2015.

[9] J. Dean and S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters,” *Proc. OSDI - Symp. Oper. Syst. Des. Implement.*, pp. 137–149, 2004.

[10] Institutions that are using Apache Hadoop for educational or production uses. [Online]. Available: https://wiki.apache.org/hadoop/PoweredBy. [Accessed: 31-Jan-2018].

[11] S. Agarwal and Z. Khanam, “Map Reduce: A Survey Paper on Recent Expansion,” *Int. J. Adv. Comput. Sci. Appl.*, vol. 6, no. 8, 2015.

[12] Z. Khanam and S. Agarwal, “Map-Reduce Implementations: Survey and Performance Comparison,” *Int. J. Comput. Sci. Inf. Technol.*, vol. 7, no. 4, pp. 119–126, 2015.

[13] Apache Hadoop: open-source software for reliable, scalable, distributed computing. [Online]. Available:

https://hadoop.apache.org/. [Accessed: 31-Jan-2018].

[14] Apache Flink: Scalable batch and stream data processing. [Online]. Available: https://flink.apache.org. [Accessed: 31-Jan-2018].

[15] A. Alexandrov *et al.*, "The Stratosphere platform for big data analytics," *VLDB J.*, vol. 23, no. 6, pp. 939–964, 2014.

[16] Apache Spark: a fast and general engine for large-scale data processing. [Online]. Available: https://spark.apache.org. [Accessed: 31-Jan-2018].

[17] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark : Cluster Computing with Working Sets," *HotCloud'10 Proc. 2nd USENIX Conf. Hot Top. cloud Comput.*, p. 10, 2010.

[18] M. C. Schatz, "CloudBurst: highly sensitive read mapping with MapReduce," *Bioinformatics*, vol. 25, no. 11, pp. 1363–1369, Jun. 2009.

[19] H. Kocakulak and T. T. Temizel, "A Hadoop solution for ballistic image analysis and recognition," in *2011 International Conference on High Performance Computing & Simulation*, 2011, pp. 836–842.

[20] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan, "An Analysis of Traces from a Production MapReduce Cluster," in *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, 2010, pp. 94–103.

[21] K. Ren, Y. Kwon, M. Balazinska, and B. Howe, "Hadoop's adolescence," *Proc. VLDB Endow.*, vol. 6, no. 10, pp. 853–864, 2013.

[22] T. Xiao *et al.*, "Nondeterminism in MapReduce considered harmful? an empirical study on non-commutative aggregators in MapReduce programs," in *Companion Proceedings of the 36th International Conference on Software Engineering - ICSE Companion 2014*, 2014, pp. 44–53.

[23] J. Moran, C. de la Riva, and J. Tuya, "MRTree: Functional Testing Based on MapReduce's Execution Behaviour," in *2014 International Conference on Future Internet of Things and Cloud*, 2014, pp. 379–384.

[24] L. C. Camargo and S. R. Vergilio, "Classifica{ç} ao de Defeitos para Programas MapReduce: Resultados de um Estudo Emp{\i}rico," 2013.

[25] J. Moran, B. Rivas, C. De La Riva, J. Tuya, I. Caballero, and M. Serrano, "Infrastructure-Aware Functional Testing of MapReduce Programs," in *2016 IEEE 4th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW)*, 2016, pp. 171–176.

[26] J. Morán, B. Rivas, C. De Riva, J. Tuya, I. Caballero, and M. Serrano, "Configuration / Infrastructure-aware testing of MapReduce programs," *Adv. Sci. Technol. Eng. Syst. J.*, vol. 2, no. 1, pp. 90–96, 2017.

[27] T. Chen, S. Cheung, and S. Yiu, "Metamorphic testing: a new approach for generating next test cases," *Tech. Rep. HKUST-CS98-01, Dep. Comput. Sci. Hong Kong Univ. Sci. Technol. Hong Kon*, pp. 1–11, 1998.

[28] Apache MRUnit: Java library that helps developers unit test Apache Hadoop map reduce job. [Online]. Available: http://mrunit.apache.org. [Accessed: 31-Jan-2018].

[29] R. L. Bocchino, V. S. Adve, S. V Adve, M. Snir, and R. L. B. Jr., "Parallel Programming Must Be Deterministic by Default," *Proc. First USENIX Conf. Hot Top. parallelism*, vol. 22, no. 1, p. 4, 2009.

[30] Average temperature per year. [Online]. Available: https://github.com/t2013anurag/Hadoop-Map-Reduce-Avg-Temp. [Accessed: 31-Jan-2018].

[31] Average temperature per year. [Online]. Available: https://github.com/hanasu/ClimateData. [Accessed: 31-Jan-2018].

[32] J. Lin and C. Dyer, "Data-Intensive Text Processing with MapReduce," *Synth. Lect. Hum. Lang. Technol.*, vol. 3, no. 1, pp. 1–177, 2010.

[33] Open Ankus: Data mining and machine learning based on MapReduce. [Online]. Available: http://www.openankus.org/.

[34] A. Orso and G. Rothermel, "Software testing: a research travelogue (2000–2014)," *Proc. Futur. Softw. Eng. - FOSE 2014*, pp. 117–132, 2014.

[35] A. Bertolino, "Software Testing Research : Achievements , Challenges , Dreams," in *Future of Software Engineering. FOSE '07*, 2007, pp. 85–103.

[36] S. Nachiyappan and S. Justus, "Getting ready for BigData testing: A practitioner's perception," in *2013 4th International Conference on Computing, Communications and Networking Technologies, ICCCNT 2013*, 2013.

[37] A. Mittal, "Trustworthiness of Big Data," *Int. J. Comput. Appl.*, vol. 80, no. 9, pp. 35–40, Oct. 2013.

[38] Z. Liu, "Research of performance test technology for big data applications," *2014 IEEE Int. Conf. Inf. Autom. ICIA 2014*, no. July, pp. 53–58, 2014.

[39] A. S. Nagdive, R. M. Tugnayat, P. Shri, S. Agnihotri, and M. P. Tembhurkar, "Overview on Performance Testing Approach in Big Data," *Int. J. Adv. Res. Comput. Sci.*, vol. 5, no. 8.

[40] M. Gudipati, S. Rao, N. D. Mohan, and N. Kumar Gajja, "Big Data: Testing Approach to Overcome Quality Challenges," vol. 11, no. 1. Big Data: Challenges and Opportunities, pp. 65–72, 2013.

[41] C. Csallner, L. Fegaras, and C. Li, "New Ideas Track: Testing Mapreduce-style Programs," *Proc. 19th ACM SIGSOFT Symp. 13th Eur. Conf. Found. Softw. Eng.*, pp. 504–507, 2011.

[42] Y.-F. Chen, C.-D. Hong, N. Sinha, and B.-Y. Wang, "Commutativity of Reducers," Springer Berlin Heidelberg, 2015, pp. 131–146.

[43] K. V. Vishwanath and N. Nagappan, "Characterizing Cloud Computing Hardware Reliability," *Proc. 1st ACM Symp. Cloud Comput. - SoCC '10*, p. 193, 2010.

[44] F. Faghri, S. Bazarbayev, M. Overholt, R. Farivar, R. H. Campbell, and W. H. Sanders, "Failure scenario as a service (FSaaS) for Hadoop clusters," *Proc. Work. Secur. Dependable Middlew. Cloud Monit. Manag. - SDMCMM '12*, pp. 1–6, 2012.

[45] P. Joshi, H. S. Gunawi, and K. Sen, "PREFAIL: A Programmable Tool for Multiple-Failure Injection," *ACM SIGPLAN Not.*, vol. 46, no. 10, p. 171, 2011.

[46] Anarchy Ape: Fault injection tool for Hadoop cluster from Yahoo anarchyape. [Online]. Available: https://github.com/david78k/anarchyape. [Accessed: 31-Jan-2018].

[47] Chaosmonkey: Fault injector. [Online]. Available: https://github.com/Netflix/SimianArmy/wiki/Chaos-Monkey. [Accessed: 31-Jan-2018].

[48] Hadoop Injection Framework. [Online]. Available: https://wiki.apache.org/hadoop/HowToUseInjectionFramework. [Accessed: 31-Jan-2018].

[49] J. E. Marynowski, A. O. Santin, and A. R. Pimentel, "Method for testing the fault tolerance of MapReduce frameworks," *Comput. Networks*, vol. 86, pp. 1–13, 2015.

[50] J. Morán, C. de la Riva, and J. Tuya, "Testing data transformations in MapReduce programs," in *Proceedings of the 6th International Workshop on Automating Test Case Design, Selection and Evaluation - A-TEST 2015*, 2015, pp. 20–25.

[51] A. J. de Mattos, "Test data generation for testing mapreduce systems," Federal University of Paraná, 2011.

[52] N. Li, Y. Lei, H. R. Khan, J. Liu, and Y. Guo, "Applying combinatorial test data generation to big data applications," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*, 2016, pp. 637–647.

[53] Herriot: Large-scale automated test framework. [Online]. Available: https://wiki.apache.org/hadoop/HowToUseSystemTestFramework. [Accessed: 31-Jan-2018].

[54] Minicluster: Apache hadoop cluster in memory for testing. [Online]. Available: https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/CLIMiniCluster.html. [Accessed: 31-Jan-2018].

[55] JUnit: a simple framework to write repeatable tests. [Online]. Available: http://junit.org. [Accessed: 31-Feb-2018].

[56] R. Hamlet, *Random testing*, no. 1. Wiley, 1994.

[57] J. M. Glenford, *The art of software testing*. 1979.

[58] M. Grindal, J. Offutt, and S. F. Andler, "Combination testing strategies: A survey," *Softw. Test. Verif. Reliab.*, vol. 15, no. 3, pp. 167–199, 2005.

[59] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Comput. Surv.*, vol. 43, no. 2, pp. 1–29, 2011.

[60] D. Hamlet and R. Taylor, "Partition Testing Does Not Inspire Confidence," *IEEE Trans. Softw. Eng.*, vol. 16, no. 12, pp. 1402–1411, 1990.

[61] J. W. Duran and S. C. Ntafos, "An Evaluation of Random Testing," *IEEE Trans. Softw. Eng.*, vol. SE-10, no. 4, pp. 438–444, 1984.

[62] A. Arcuri, M. Z. Iqbal, and L. Briand, "Random testing: Theoretical results and practical implications," *IEEE Trans. Softw. Eng.*, vol. 38, no. 2, pp. 258–277, 2012.

[63] W. J. Gutjahr, "Partition testing vs. random testing: The influence of uncertainty," *IEEE Trans. Softw. Eng.*, vol. 25, no. 5, pp. 661–674, 1999.

[64] ISO/IEC/IEEE, "29119-4:2015 - ISO/IEC/IEEE International Standard for Software and systems engineering — Software testing —

Part 4: Test techniques,” *ISO/IEC/IEEE 29119-4:2015*, pp. 1–149, 2015.

[65] A. W. Williams and R. L. Probert, “A measure for component interaction test coverage,” in *Proceedings of IEEE/ACS International Conference on Computer Systems and Applications, AICCSA*, 2001, vol. 2001–Janua, pp. 304–311.

[66] P. Ammann and J. Offutt, “Using formal methods to derive test frames in category-partition testing,” *Comput. Assur. 1994. COMPASS ’94 Safety, Reliab. Fault Toler. Concurr. Real Time, Secur. Proc. Ninth Annu. Conf.*, pp. 69–79, 1994.

[67] D. R. Kuhn and M. J. Reilly, “An investigation of the applicability of design of experiments to software testing,” in *Proceedings - 27th Annual NASA Goddard / IEEE Software Engineering Workshop, SEW 2002*, 2003, pp. 91–95.

[68] J. Huller, “Reducing time to market with combinatorial design method testing,” in *IN PROCEEDINGS OF THE 2000 INTERNATIONAL COUNCIL ON SYSTEMS ENGINEERING (INCOSE) CONFERENCE*, 2000, pp. 16--20.

[69] E. J. Weyuker, “On testing non-testable programs,” *Comput. J.*, vol. 25, no. 4, pp. 465–470, 1982.

[70] M. Staats, M. W. Whalen, and M. P. E. Heimdahl, “Programs, tests, and oracles: the foundations of testing revisited,” in *Proceeding of the 33rd international conference on Software engineering - ICSE ’11*, 2011, p. 391.

[71] R. A. P. Oliveira, U. Kanewala, and P. A. Nardi, “Automated test oracles: State of the art, taxonomies, and trends,” *Adv. Comput.*, vol. 95, pp. 113–199, 2015.

[72] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, “The oracle problem in software testing: A survey,” *IEEE Trans. Softw. Eng.*, vol. 41, no. 5, pp. 507–525, 2015.

[73] ISO/IEC/IEEE, “29119-1:2013 - ISO/IEC/IEEE International Standard for Software and systems engineering — Software testing — Part 1: Concepts and definitions,” *ISO/IEC/IEEE 29119-1:2013(E)*, vol. 2013, pp. 1–64, 2013.

[74] X. Xie, J. W. K. Ho, C. Murphy, G. Kaiser, B. Xu, and T. Y. Chen, “Testing and validating machine learning classifiers by metamorphic testing,” in *Journal of Systems and Software*, 2011, vol. 84, no. 4, pp. 544–558.

[75] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortes, “A Survey on Metamorphic Testing,” *IEEE Trans. Softw. Eng.*, vol. 42, no. 9, pp. 805–824, Sep. 2016.

[76] R. W. Selby and V. R. Basili, “Comparing the Effectiveness of Software Testing Strategies,” *IEEE Trans. Softw. Eng.*, vol. SE-13, no. 12, pp. 1278–1296, 1987.

[77] H. A. De Souza, M. L. Chaim, and F. Kon, “Spectrum-based Software Fault Localization: A Survey of Techniques, Advances, and Challenges,” *arxiv16*, pp. 1–40, 2016.

[78] S. B. Kotsiantis, “Supervised Machine Learning: A Review of Classification Techniques,” *Informatica*, vol. 31, pp. 249–268, 2007.

[79] V. R. Basili and H. Dieter Rombach, “The TAME Project: Towards Improvement-Oriented Software Environments,” *IEEE Trans. Softw. Eng.*, vol. 14, no. 6, pp. 758–773, 1988.

[80] B. Rivas, J. Merino, M. Serrano, I. Caballero, and M. Piattini, “I8K|DQ-BigData: I8K architecture extension for data quality in big data,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2015, vol. 9382, pp. 164–172.

[81] “Movies analysis implemented in MapReduce.” [Online]. Available: https://github.com/adityaundirwadkar/mapreduce-programming/tree/master/example_1. [Accessed: 31-Jan-2018].

[82] Treelogic S.L. [Online]. Available: www.treelogic.com. [Accessed: 31-Jan-2018].

[83] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*, vol. 9783642290. 2012.

[84] MapReduce algorithm of Connected components in graphs. [Online]. Available: https://github.com/Draxent/ConnectedComponents. [Accessed: 31-Jan-2018].

[85] Goldstein analysis implemented in MapReduce. [Online]. Available: https://github.com/tchira/MapReduce/tree/master/src/main/java/hadoop/gratio. [Accessed: 31-Jan-2018].

[86] Analysis of the New York restaurants based on MapReduce. [Online]. Available: https://github.com/Shubham617/MapReduce-Project/tree/master/NYC Restaurant Data. [Accessed: 31-Jan-2018].

[87] T. D. Cook and D. T. (Donald T. Campbell, *Quasi-experimentation : design &amp; analysis issues for field settings*. Houghton Mifflin, 1979.

[88] R. Malhotra, *Empirical research in software engineering : concepts, analysis, and applications*. .

[89] A. S. Namin and S. Kakarla, “The use of mutation in testing experiments and its sensitivity to external threats,” *Proc. 2011 Int. Symp. Softw. Test. Anal. - ISSTA ’11*, p. 342, 2011.

[90] M. Papadakis, C. Henard, M. Harman, Y. Jia, and Y. Le Traon, “Threats to the validity of mutation-based test assessment,” in *Proceedings of the 25th International Symposium on Software Testing and Analysis - ISSTA 2016*, 2016, pp. 354–365.

[91] J. H. Andrews, L. C. Briand, and Y. Labiche, “Is mutation an appropriate tool for testing experiments?,” in *Proceedings of the 27th international conference on Software engineering - ICSE ’05*, 2005, p. 402.

[92] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, “Are mutants a valid substitute for real faults in software testing?,” in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*, 2014, pp. 654–665.

[93] R. Guderlei and J. Mayer, “Statistical Metamorphic Testing - Testing programs with random output by means of statistical hypothesis tests and Metamorphic Testing,” in *Proceedings - International Conference on Quality Software*, 2007, pp. 404–409.

[94] C. Murphy and G. Kaiser, “Empirical evaluation of approaches to testing applications without test oracles,” *Dep. Comput. Sci. Columbia Univ. Tech. Rep. CUCS-039-09*, 2010.

Jesús Morán received his B.Sc. degree in Computer Engineering in 2012 and an M.Sc. in Computer Engineering in 2014 from the University of Oviedo, Spain. He is currently a PhD candidate at the University of Oviedo. His research interests include software testing, Big Data technologies and distributed programming.

Antonia Bertolino is a research director at CNR-ISTI (Italian National Research Council—Institute of Information Science and Technology), Pisa, Italy. Her research focuses on software and service testing. Bertolino received an MS in Electronic Engineering from the University of Pisa. She is an associate editor of ACM Transactions on Software Engineering and Methodology and of Springer Empirical Software Engineering Journal, and serves as the Software Testing area editor of the Elsevier Journal of Systems and Software. She has been the General Chair of the 2015 International Conference on Software Engineering held in Florence (Italy).

Claudio de la Riva is an Assistant Professor at the University of Oviedo. He is a member of the Software Engineering Research Group (GIIS, giis.uniovi.es). He obtained his PhD in Computing from the University of Oviedo. His research interests include software verification and validation and software testing, mainly focused on testing database applications and services. He is a member of ACM.

Javier Tuya is a Professor at the University of Oviedo, Spain, where he is the research leader of the Software Engineering Research Group. He received his PhD in Engineering from the University of Oviedo in 1995. He is the Director of the Indra-Uniovi Chair, member of the ISO/IEC JTC1/SC7/WG26 working group for the recent ISO/IEC/IEEE 29119 Software Testing standard and convener of the corresponding UNE National Body working group. His research interests in software engineering include verification & validation and software testing for database applications and services. He is a member of the IEEE, IEEE Computer Society, ACM and the Association for Software Testing (AST).