A Method-Level Test Generation Framework for Debugging Big Data Applications

Huadong Feng, Jaganmohan Chandrasekaran, Yu Lei
Department of Computer Science & Engineering
The University of Texas at Arlington
Arlington, USA
{huadong.feng, jaganmohan.chandrasekaran}@mavs.uta.edu,
ylei@cse.uta.edu

Raghu Kacker, D. Richard Kuhn
Information Technology Lab
National Institute of Standards and Technology
Gaithersburg, USA
{raghu.kacker, d.khun}@nist.gov

***Abstract*—When a failure occurs in a big data application, debugging with the original dataset can be difficult due to the large amount of data being processed. This paper introduces a framework for effectively generating method-level tests to facilitate debugging of big data applications. This is achieved by running a big data application with the original dataset and by recording the inputs to a small number of method executions, which we refer to as method-level tests, that preserve certain code coverage, e.g., edge coverage. The size of each method-level test is further reduced if needed, while maintaining code coverage. When debugging, a developer could inspect the execution of these method-level tests, instead of the entire program execution with the original dataset. We applied the framework to seven algorithms in the WEKA tool. The initial results show that in many cases a small number of method-level tests are sufficient to preserve code coverage. Furthermore, these tests could kill between 57.58% to 91.43% of the mutants generated using a mutation testing tool. This suggests that the framework could significantly reduce the efforts required for debugging big data applications.**

Keywords—Testing; Unit Testing; Big Data Application Testing; Test Generation; Test Reduction; Debugging; Mutation Testing;

I. INTRODUCTION

Big data applications are software programs that process large amounts of data. Debugging big data applications can be complicated and time-consuming. This is due to the fact that inspecting the execution of a big data application often involves long execution time, a large number of method executions, and/or a large number of objects. For example, a classification algorithm, called *DecisionTable*, in the WEKA tool [12] takes more than two hours to execute the *Heterogeneity Activity Recognition Dataset* (HAR) from the UC Irvine (UCI) Machine Learning Repository [13]. During the execution, one of the *DecisionTable's* methods, named *updateStatsForClassifier*, is executed more than half a billion times. (This method has 66 lines of code, not including comments and spaces.) If there exists a fault in this method, it can be very difficult to locate this fault due to the large number of times this method is executed.

Some approaches have been proposed to reduce the effort required for testing and debugging big data applications at the system level [1, 2, 3, 4, 5]. For example, data mining and machine learning methods are used to reduce the size of the original dataset or generate synthetic datasets [3, 4] for the testing purpose. The reduced dataset using such methods are executed at the system level, which can still be time-consuming. Furthermore, these methods are not designed to reproduce the original failure. Debugging approaches such as delta debugging [8] can identify the minimum failure-inducing input at the system level, which can reduce the size of the input while preserving the failure triggered by the original dataset. However, delta debugging can be very expensive for big data applications. This is because it requires the input data be recursively split into smaller chunks, each of which has to be executed at the system level. For big data applications, there can be a large number of chunks and system-level execution of each chunk can be time-consuming.

Our approach consists of two major steps. In the first step, we re-execute the failing system-level execution to record method-level tests for suspicious method(s). The main idea is to evaluate each method execution based on a chosen coverage criterion. In this paper, we used edge coverage, edge-pair coverage and edge-set coverage based on the Control Flow Graph (CFG) [11]. Note that other coverage criteria, e.g., prime-path coverage [11], could also be used in our approach. We record the input to a method execution as a method-level test when it covers any new coverage element with respect to the chosen coverage criterion. In the second step, we reduce method-level tests with large collection-typed variables using binary reduction. The reduced tests preserve the same coverage achieved by the originally recorded method-level tests. During debugging, a developer will first identify suspicious methods based on his or her understanding of the program. Then, the developer will only need to re-execute the reduced method-level tests recorded for these methods, instead of executing the entire application with the original dataset. Doing so could significantly speed up the debugging process.

We conducted an experimental evaluation of our approach. In our experiments, we selected seven methods from four machine learning algorithms that were implemented in WEKA using Java. The four machine learning algorithms from WEKA and two datasets from UCI dataset repository were selected based on the execution time and size of datasets. Method-level tests were recorded for these seven methods based on three coverage criteria, including edge coverage, edge-pair coverage, and edge-set coverage. (The three coverage criteria are defined

978-1-5386-5035-6/18/$31.00 ©2018 IEEE

in Section II-A.) On average, 4.4 tests were recorded for edge coverage, 5.9 tests for edge-pair coverage, and 18.6 tests for edge-set coverage. While initially, the seven methods were executed from 191 to half a billion times. For some of the recorded method-level tests with large-size inputs, e.g., the previously mentioned *updateStatsForClassifier* method in the *DecisionTable* algorithm, we further reduced the size of the inputs using a binary reduction technique while preserving the same coverage achieved by the original method-level test. For example, the average input size for *updateStatsForClassifier* was reduced to 12.53 MB from 1269.76 GB.

Moreover, test effectiveness was evaluated using PITest (PIT) [16], a commonly used mutation testing tool. Mutation testing seeds faults in a systematic manner to simulate mistakes that developers may make during programming. All 25 available mutant generators were enabled for mutant generation. When combining each set of tests generated for the edge, edge-pair, and edge-set coverage for each method, the mutant killing rate ranges from 57.58% to 91.43%.

We summarize the contributions of our paper as follows:

- We present a new framework for debugging big data applications based on method-level tests. Compared to executing the original dataset at the system level, these method-level tests can be much faster to execute and inspect, which could significantly speed up the debugging process.

- We built a prototype that implements our framework and conducted an experimental evaluation of the framework. The evaluation results suggest that our framework could significantly reduce the time and effort required for debugging big data applications.

The rest of the paper is organized as follows. Section II presents the details of our approach and discusses several implementation challenges. Section III presents the experimental design and analysis of the experimental results. Section IV provides an overview of existing work that is closely related to ours. Section V provides concluding remarks as well as several directions for our future work.

II. Approach

Our approach consists of two major steps, recording method-level tests and reducing the size of the recorded tests. In this section, Section II-A presents our approach to recording method-level tests based on a given coverage criterion. Section II-B presents our approach to reducing the size of a recorded test.

A. Record Test

In a typical scenario, once a failure occurs, a developer identifies several suspicious locations based on his or her understanding of the program. Next, the developer could set up breakpoints in these locations and then start the debugging process with the system-level inputs. The breakpoints allow the developer to inspect the program state during the debugging process. This approach may not be effective for big data applications. This is because when the dataset is large, a breakpoint may be executed for a large number of times before an incorrect program state is found, and each breakpoint has to be inspected manually.

In our approach, the developer first identifies suspicious methods, in a way that is similar to the identification of suspicious locations. Next, our approach runs the program with the original dataset and records, for each suspicious method, a small number of method executions, which we refer to as method-level tests, based on a specific coverage criterion. The method-level tests recorded for a given method achieve the same coverage criterion as the original dataset for the method. The developer can then debug each method with the recorded method executions, instead of a potentially large number of method executions. Since the same coverage criterion is satisfied, there is a high probability that debugging these recorded method-level tests would allow us to detect the fault that may have caused the failure observed at the system level.

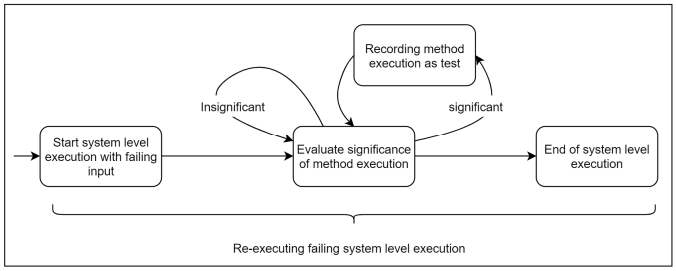


Fig. 1. Recording Process at Runtime

After the developer identifies a list of suspicious methods to be recorded, we instrument these methods to capture the coverage elements that need to be covered for the selected coverage criterion. After instrumentation, our recording process at runtime is shown in Figure 1. While re-executing the failing system-level execution, each method execution of the suspicious methods is evaluated to determine whether it is significant based on the selected coverage criterion. A method execution is considered to be significant if it covers at least one new coverage element. When a method execution is deemed to be significant, its corresponding input for reproducing the method execution is recorded as a method-level test. Otherwise, the execution will continue until it reaches the next significant method execution.

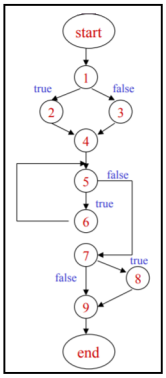
In this paper, we will use edge coverage [11], edge-pair coverage [11], and edge-set coverage, as the coverage criteria based on Control Flow Graph (CFG) to determine if a given method execution is significant. A CFG is a graphical representation of all possible paths that might be traversed by a program at runtime. Thus it captures information about how the control is transferred in a program.

Figure 2 shows an example CFG. In a CFG, each node in the graph represents a basic block, i.e. a sequence of consecutive statements with a single entry and a single exit point[11]. A directed edge [11] represents that the control can flow from one node to another. And a path [11] is a sequence of nodes, where each pair of adjacent nodes is an edge.

We record the method executions as method-level tests when they cover any new coverage elements with respect to

the chosen coverage criterion. For edge coverage, each edge covered by a method execution is recorded for the method evaluation. For edge-pair coverage, each edge-pair (reachable path of length up to two) is recorded for the method evaluation. Note that when edge-set coverage is used, a method execution is considered significant if it covers a unique set of edges, i.e., no other method executions exactly cover the same set of edges. Also note that other coverage criteria, e.g., prime-path coverage [11], could also be used in our approach.

```
1.  begin
2.    int x, y, power;
3.    float z;
4.    input (x, y);
5.    if (y < 0)
6.       power = -y;
7.    else
8.       power = y;
9.    z = 1;
10.   while (power != 0) {
11.      z = z * x;
12.      power = power – 1;
13.   }
14.   if (y < 0)
15.      z = 1/z;
16.   output (z);
17. end;
```



Block	Lines	Entry	Exit
1	2, 3, 4, 5	2	5
2	6	6	6
3	8	8	8
4	9	9	9
5	10	10	10
6	11, 12	11	12
7	14	14	14
8	15	15	15
9	16	16	16

Fig. 2. Example of Control Flow Graph

To record method-level tests, three major tasks need to be accomplished, including instrumentation, method execution evaluation, and serialization. We further discuss these tasks in the following subsections.

1) Instrumentation

We use a tool called Atlas [15], which is an Eclipse plugin developed by EnSoft Corp to automatically generate CFGs from the source code of a selected method. Atlas uses each line of code as a basic block. This is different from the classical definition [11] that a basic block consists of a sequence of consecutive statements with a single entry and a single exit point. Figure 2 shows a simple method and its CFG generated using Atlas. We modify the generated CFGs from Atlas by combining blocks that are in a consecutive sequence without inner branches. Doing so reduces the amount of instrumentation and thus the runtime overhead when executing the instrumented code. The red rectangle in Figure 3 marks the lines of code combined to be a basic block as we previously defined.

```
public float power(int x, int y){
    int power;
    float z;
    if (y < 0){
        power = -y;
    }else{
        power = y;
    }
    z = 1;
    while (power != 0){
        z = z * x;
        power = power - 1;
    }
    if(y < 0){
        z = 1 / z;
    }
    return z;
}
```

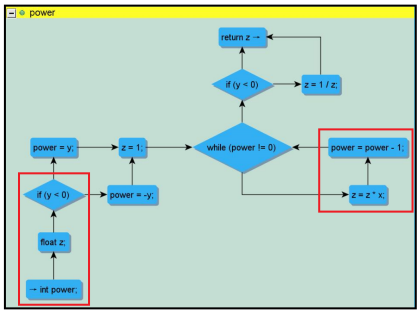


Fig. 3. Example of Modifying Generated Control Flow Graph

Once we have the CFG of a suspicious method, we instrument the method by adding a few lines of code that invokes our recording program. Figure 4 shows an example of how we instrument a sample method. The highlighted statements are extra code added by instrumentation. The code from line 3 to line 10 initializes the recording process. They are inserted at the beginning of a suspicious method. The *ParaArray* array contains the list of input parameters used for a method execution. The *ParaTypeArray* array contains the object types of the input parameters, which are needed to reload the recorded inputs using Java Reflection. When recording a method execution, we record not only the input parameters but also the current object on which the suspicious method was invoked, to store the instance variables accessed during the execution. They are loaded into our system using the "*R.loadInputs(ParaArray, this);*" statement. The statement "*R.enterBlock(#number);*" is added before each basic block to record the index of the basic block when it is executed. The block number *#number* is manually determined based on the previously discussed CFG. Moreover, the statement "*R.endOfProcess();*" is added before each return statement or at the end of a method to notify our program a method execution is completed, and start the method execution evaluation process.

```
1  public float power(int x, int y) {
2      //Insert @ Beginning Of Method
3      RecordMethodExecution R = new RecordMethodExecution();
4      Object[] ParaArray = {x, y};
5      Class[] ParaTypeArray =
6              new Object() {}.getClass().getEnclosingMethod().getParameterTypes();
7      String MethodName = this.getClass().getName() + "/" +
8              new Object() {}.getClass().getEnclosingMethod().getName();
9      R.initializeProcess(MethodName, ParaTypeArray);
10     R.loadInputs(ParaArray, this);
11     //Insert @ Beginning Of Each Block
12     R.enterBlock(0);
13     int power;
14     float z;
15     if (y < 0) {
16         R.enterBlock(1);
17         power = -y;
18     }else {
19         R.enterBlock(2);
20         power = y;
21     }
22     R.enterBlock(3);
23     z = 1;
24     while (power != 0) {
25         R.enterBlock(4);
26         R.enterBlock(5);
27         z = z * x;
28         power = power - 1;
29     }
30     R.enterBlock(4);
31     R.enterBlock(6);
32     if (y < 0) {
33         R.enterBlock(7);
34         z = 1 / z;
35     }
36     R.enterBlock(8);
37     //Insert @ Before Each Return or End of Method
38     R.endOfProcess();
39     return z;
40 }
```

Fig. 4. Example of Instrumentation

Recording basic block indexes with multiple entrances at runtime requires more work than just adding the "*R.enterBlock(#number)"* statement in front of it. As shown in Figure 4, lines 25 to 26 and lines 30 to 31 are the extra codes added for recording the basic block contains line 24. To record the basic block indexes correctly for basic blocks with multiple entrances such as for *while* loop, *for* loop, *else if*, and *switch* statements, etc., we are inserting the "*R.enterBlock(#number);*" statement before its descendants' "*R.enterBlock(#number);*" statement based on the CFG to capture every execution of such blocks. For example, if we only add "*R.enterBlock(#number);*"

statement right before the *while* statement shown in Figure 4 at line 24, when the loop comes back to re-evaluate the loop condition at the *while* statement, the repeated execution of this block will not be captured.

2) Method Execution Evaluation

In our implemented framework, we temporarily store the covered edges, edge-pairs, and edge-set for each method execution. We consider a method execution to be significant, and thus record the execution as a method-level test if it covers any edge, edge-pair or edge-set that has not been covered before. Note that we check for uncovered edges first for each method execution. This is because if a method execution covers any edge that has not been covered before, it must cover some new edge-pair(s) and a new edge-set. The time complexity for evaluating each method executions is $O(n^2)$ where n represents the number of coverage elements each method execution has to evaluate. For each method execution, each coverage element of the method execution will be compared to the list of the previously covered elements. If a method execution covers any new coverage element, the method execution will be recorded, and the newly covered elements will be added to the list.

3) Serialization

Once a method execution is determined to be significant, we record the inputs of the method execution using serialization. Serialization can be an expensive process, the built-in serialization support in Java is rather slow when serializing large objects. We used an alternative tool called FST [17] that can be ten times faster [17] to improve the performance of our test recording. In our experiments, FST was able to serialize and deserialize objects correctly. However, there are some reported cases [17] where FST was unable to correctly serialize and deserialize objects that the built-in Java serialization could. In comparison, FST provides better performance, but FST does not provide serialization ability that is as strong as the Java built-in serialization.

While our performance is improved using FST, there are still some situations where we experience significant overhead. To ensure an exact copy of the input objects is created, we perform deep copy on the objects by serializing and deserializing these objects. This is needed because the value of an input object could potentially change during a method execution, especially for void methods that operate on instance variables.

However, most of the stored input objects will not be recorded if the method execution does not cover any new coverage element. Thus, much of the time spent to store the deep copies of objects is unnecessary. These unnecessary time can be huge when a method takes large inputs and/or is executed for a large number of times. The recording overhead can be as high as 7 to 30 times the original system-level execution time for some of the selected methods. In such cases, our solution is recording the method-level tests by executing the entire system twice. In the first execution, we do not store any inputs. Instead, we only record the IDs of significant method executions. In the second execution, we only serialize the selected method executions to store their inputs as method-level tests. Doing so can significantly reduce the runtime overhead in cases where a method takes large inputs or is being executed for a large number of times.

B. Test Reduction

While the recorded method-level tests can be used for debugging, these tests in some case consist of very large inputs. For example, one of the selected methods *cutPointsForSubset,* its recorded method-level tests have the average size of 1.62GB, executing these tests can take a lot of time. And breakpoints in loop statements can be executed for a large number of times. These inputs are large mostly due to the fact that they contain large collections of objects. For the three methods mentioned above, they all have *Instances* typed (Implements *Collection*) variables that contain instances from the original dataset for processing. Some of the recorded data could potentially be reduced while still reproducing the method execution and preserving the coverage elements. The reduction can further reduce the time for executing the tests, and the debugging efforts required from developers.

Our binary reduction technique is inspired by the commonly used binary search technique. For each recorded method-level test, we divide its collection typed input variables into halves. Next, we take each half and other non-collection typed inputs and re-execute them with the suspicious method. We then check whether a half can preserve the originally covered coverage elements. If one of the halves does preserve all the coverage elements, we will continue dividing it into halves and check for the coverage elements repeatedly, until the minimal subset of the collection variables that can preserve the coverage elements are identified. Note that when preserving the coverage during reduction, we are preserving the exact covered elements of edge coverage, edge-pair coverage, and edge-set coverage.

III. EXPERIMENTS

We implemented the initial working prototype of our framework in Java. Some Manual efforts are required from developers to instrument the source code of suspicious methods. After instrumentation, the recording process has been automated. The reduction approach requires developers to manually identify the large collection typed input variables. The re-execution of the recorded and reduced method-level tests has been automated for debugging. We also conducted mutation testing to evaluate the fault detection effectiveness of our recorded and reduced method-level test. The currently implemented coverage criteria are the edge, edge-pair, and edge-set coverage.

In the following, we discuss how we conducted our experiments and present the experiment results. In Section III-A, we discuss how we selected datasets, applications, and methods to be used for our experiments. Section III-B presents the statistics of the recorded method-level tests. Section III-C presents the statistics of the reduced method-level tests. Section III-D presents how we conducted a mutation testing experiment and the results of our mutation testing for both the recorded tests, and the reduced tests. And finally, Section III-E presents the performance analysis of our framework. All the

source code, recorded method-level tests, reduced method-level tests and mutation reports are publicly available at https://www.dropbox.com/sh/3k4kjwqjpa9i2qv/AAAkeYYNaQOVfT9WGe4OUp_Pa?dl=0 for review. The machine we used for our experiment is a workstation with two Xeon E5-2630V3 8 core CPUs @ 2.40GHz, 64GB DDR4 2133 MT/s memory, and a Samsung 850 EVO 500GB SSD.

A. Subjects

We design our experiments to reflect real-world situations for evaluating the effectiveness of our framework. First, we randomly selected ten algorithms that are implemented in the WEKA tool. WEKA is one of the most widely used tools for data mining by practitioners. Next, we selected one collection of dataset with the largest number of instances (accessed on 08/18/2018) from the UCI Machine Learning Repository that consists of 440 real-world collected datasets as a start. The selected collection of datasets, *Heterogeneity Activity Recognition* (HAR), contains four datasets for four different types of devices with a total of 43,930,257 instances and 16 attributes. The HAR collection includes several data types, including multivariate, time-series and real numbers. The datasets can be used for both classification and clustering. Among the four datasets, the largest dataset, *Phones_gyroscope*, is used to execute the ten algorithms.

Phones_gyroscope dataset has the size of 1.37GB, it is too large for two of our selected algorithms *EM* and *LibSVM* to finish their execution within a day. The execution time is too long for our experimentation purpose due to our limited time and resources. For these two algorithms, we reduced the size of the *Phones_gyroscope* dataset by dividing the dataset in half and continue to divide in half until the execution time for *EM* and *LibSVM* are reduced to be near an hour. The reduced *Phones_gyroscope* dataset for *EM* and *LibSVM* now has the size of 3.3 MB. *EM* will now take 5352 seconds (1.49 Hours) to execute and 4491 seconds (1.25 Hours) for *LibSVM*.

TABLE I. SELECTED METHOD INFORMATION

Method	Algorithm	# of Covered Lines of Code	# of Total Lines of Code	# of Execution Count
buildClusterer	EM	115	165	1,910
cutPointsForSubset	DecisionTable	62	64	29,564
EM_Init	EM	47	53	191
handleNumericAttribute	J48	51	53	28,314
select_working_set	LibSVM	50	52	417,989
selectModel	J48	50	58	12,391
updateStatsForClassifier	DecisionTable	46	66	557,305,280

After two datasets (original *Phone_gyroscope* dataset and the reduced dataset) and ten algorithms' implementations (*Apriori, DecisionTable, EM, HierarchicalClusterer, J48, LibSVM, LinearRegression, MakeDensityBasedClusterer, RandomTree, SimpleKMeans*) have been selected. We select methods with a larger number of executed statements, and a larger number of executions for our experiments. This is because longer methods and methods that have been executed for a larger number of times often require more effort to debug. A total of seven methods are selected. The selected methods and their information are shown in Table I. These methods are then instrumented as previously described in Section II.

B. Recorded Method-Level Tests

For our experiments, we have recorded method-level tests for all of the seven selected methods for preserving edge coverage, edge-pair coverage, and edge-set coverage of the original system-level execution. Some important information about the recorded method-level tests is shown in Table II. Note that the statement coverage column in Table II is for all three types of recorded tests, as well as the original failing system-level execution. This is because edge coverage subsumes statement coverage, once all edges are preserved, all the statement coverage will be preserved as well, and edge-pair coverage and edge-set coverage both subsume edge coverage.

Based on the results shown in Table II, we can see that only a small number of method-level tests are sufficient for preserving coverage for a suspicious method. Empirical studies show that there exists a high correlation between code coverage and fault detection effectiveness. The actual fault detection ability of our recorded method-level tests will be further evaluated using mutation testing in Section III-D. Thus, when failures occur on a system level, it is likely that executing the method-level tests for the suspicious methods would trigger the failure observed during the execution with the original dataset. Thus, the use of method-level tests could potentially save developers a lot of time and efforts.

C. Reduced Method-Level Tests

As shown in Table III, while some of the tests have a reasonable size, three methods, *cutPointsForSubset*, *selectModel* and *updateStatsForClassifier* have significantly large inputs for their recorded method-level tests. While debugging with these tests is easier than debugging with the original dataset at the system level, loading and debugging these tests could still take a lot of time. We further reduce the size of these tests using our binary reduction approach as discussed in Section II. In Table III, we compare the differences between the recorded method-level tests before and after they were reduced.

For size reduction, our binary reduction technique was able to reduce the input size of tests for five out of seven methods. Our result shows that the reduction amount is often above 95%. Most of the method-level tests can be reduced significantly while still preserving our selected coverage elements. The coverage element refers to the edges, edge-pairs, and edge-set covered by each recorded method-level test. While one of the tests for *selectModel* can be reduced to 1.7 KB from 1.63 GB, some tests still have a fair amount of input data remaining, such as the reduction from 1.63GB to 37.22 MB for one of the

TABLE II. RECORDED METHOD EXECUTION INFORMATION

Method	Edge Coverage		Edge-Pair Coverage		Edge-Set Coverage		Total # of Recorded Tests	# of Original Execution Count	Statement Coverage
	# of Tests	# of Covered Edges	# of Tests	# of Covered Edge-Pairs	# of Tests	# of Covered Edge-Sets			
buildClusterer	3	83	4	181	3	3	4	1,910	69.70%
cutPointsForSubset	8	30	9	64	17	17	18	29,564	96.88%
EM_Init	1	24	3	55	1	1	3	191	88.68%
handleNumericAttribute	4	32	5	70	33	33	33	28,314	96.23%
select_working_set	7	41	11	111	61	61	63	417,989	96.15%
selectModel	5	35	5	74	6	6	6	12,391	86.21%
updateStatsForClassifier	3	26	5	59	9	9	11	557,305,280	69.70%

tests of *cutPointsForSubset*. Furthermore, we were unable to reduce any test inputs for two methods, *buildClusterer* and *EM_Init*. We further investigated this by looking into how the variables of collection type are accessed and used. We noticed mainly three different scenarios that may have contributed to our results.

The first scenario is when a collection variable is partially used as inputs. When the partially accessed instances are in a consecutive sequence in the collection variable, or when only one instance is accessed, our binary reduction technique will reduce such collection variable to its minimal subset. However, if the accessed instances are spread across the collection variable, our binary reduction will not be able to identify only the accessed instances. Hence, the reduction may not be minimal, many unnecessary data based on the coverage elements may remain.

```
744       if (m_executionSlots <= 1
745         || instances.numInstances() < 2 * m_executionSlots) {
746         for (i = 0; i < instances.numInstances(); i++) {
747           Instance toCluster = instances.instance(i);
748           int newC =
749             clusterProcessedInstance(
750               toCluster,
751               false,
752               true,
753               m_speedUpDistanceCompWithCanopies ? m_dataPointCanopyAssignments
754                 .get(i) : null);
755           if (newC != clusterAssignments[i]) {
756             converged = false;
757           }
758           clusterAssignments[i] = newC;
759         }
760       } else {
761         converged = launchAssignToClusters(instances, clusterAssignments);
762       }
```

Fig. 5. Collection Variable Used at Branching Condition

The second scenario is when the collection variable is accessed in branching statements, e.g. for the tests recorded for *buildClusterer* and *EM_Init*. The collection variables identified for these two methods were used at a few branching statements and passed to other methods that return value to the execution as well. In this situation, maintaining the exact coverage elements can be difficult to achieve for our binary reduction technique. As an example, part of the code of *buildClusterer* is shown in Figure 5. The *instances* variable was used at an *if* statement and in the conditions of a *for* loop. Reducing the *instance* variable using our binary reduction approach will compromise the originally covered coverage elements (edges, edge-pairs, edge-set) of the method-level tests recorded for the *buildClusterer* method.

The third scenario is when the collection variable is not accessed at all. In our implementation, to reduce manual efforts required for instrumentation and reproduce method executions precisely, we automatically record both the parameters passed to the method and the object where the method was invoked from, ensuring all possible inputs are recorded. However, not all recorded information is used as inputs, such as for some instance variables of the object where the method was invoked from. In this situation, our binary reduction technique may be able to reduce unnecessary collection variables to empty, while still preserving the coverage elements.

The first and second scenario can potentially use delta debugging [8] or preserving superset of the coverage elements to further the reduction. However, delta debugging could significantly increase the reduction overhead, and preserving superset of the coverage elements may lose or introduce some coverage elements that could potentially have a large impact on the reduced method-level test. For the third scenario, we can implement systematic static analysis in the future to help our framework identify and record only the necessary inputs for reproducing method executions.

For execution time reduction, many of the recorded set of method-level tests are now taking seconds instead of minutes after the binary reduction. When debugging with these reduced tests, not only the tests will be short and easier to debug, the execution time is also easy to manage.

D. Mutation Testing

For mutation testing, we used PITest (PIT) [16], a mutation testing tool for Java, to evaluate the fault detection effectiveness of our recorded method-level tests. In PIT, different types of faults (or mutants) are automatically seeded

TABLE III. TEST REDUCTION RESULTS

Method	Total # of Tests	# of Large Collection Variables	Average Input Size (MB)		Total Input Size (MB)		Maximum Input Size (MB)		Minimum Input Size (MB)		Test Execution Time (Seconds)	
			Recorded	*Reduced*	*Recorded*	*Reduced*	*Recorded*	*Reduced*	*Recorded*	*Reduced*	*Recorded*	*Reduced*
buildClusterer	4	1	3.18	3.18	12.75	12.75	3.23	3.23	3.16	3.16	5 s	5 s
cutPointsForSubset	18	2	1628.16	9.77	29306.81	176.25	1628.16	37.22	1628.16	0.001	1836 s	5 s
EM_Init	3	1	4.71	4.71	14.12	14.12	4.34	4.34	5.18	5.18	5 s	5 s
handleNumericAttribute	33	1	54.00	0.74	1781.76	24.42	1300.48	1.75	0.0012	0.001	155 s	2 s
select_working_set	63	2	39.50	0.08	2488.32	5.04	41.9	0.29	1.83	0.002	176 s	1 s
selectModel	6	2	1392.64	0.02	8357.04	0.11	1628.16	0.01	1320.96	0.001	682 s	1 s
updateStatsForClassifier	11	1	1269.76	12.53	13967.34	138.06	1269.76	44.86	1269.76	0.51	875 s	3 s

into the source code. Each mutation (a mutated version of source code) simulates a single fault and is executed against the unit tests that developers provide.

Mutation testing requires the provided unit tests to be passing tests. This is because only when the mutant's output differs from the expected output, a mutant is said to be killed. In our experiments, when a method-level test is executed, we record the outputs as the expected output for mutation testing purpose. The output for each test contains not only the returned object if there is one, but also the object where the method was invoked from and the input parameters of the method. This is because the values of these parameters and the object where the method was invoked from could change and should be considered as part of the output.

PIT provides a total of 25 different mutators to mutate different type of code. When conducting mutation testing, we have enabled all 25 mutators in PIT for generating mutants in our selected methods. PIT also provides an option to set a timeout factor for executing each test against each mutant. The default is 1.25 times the original test execution time. We increased the timeout factor to 10 times the original execution time, as an effort to avoid false positives killing of mutants. This is because a timed-out mutant is also considered as a killed mutant. We have also increased the Java heap size to 60GB and stack size to 128MB using JVM configuration in PIT, to avoid false positive killing of memory error mutants.

Table IV shows the mutation testing result of our recorded and reduced method-level tests. Note that PIT currently does not support the mutant generation of only covered statements. Because the mutation generation of PIT is done statically, it will generate mutants for all the statements of a selected method, instead of only the reachable ones. In other words, if a mutant is located at a statement that was not covered by any of the tests, the mutant will not be exercised, and thus is impossible to be killed. Such mutants will not be considered in our experiments. This is because if a mutant is not exercised by our recorded tests, it is not exercised by the original system-level execution. The total number of mutants generated for each selected method in Table IV are calculated manually which consist of only exercised mutants by our tests. This is done by removing mutants that are labeled as *NO_COVERAGE* in the mutation testing report generated using PIT, such as shown in Figure 6.

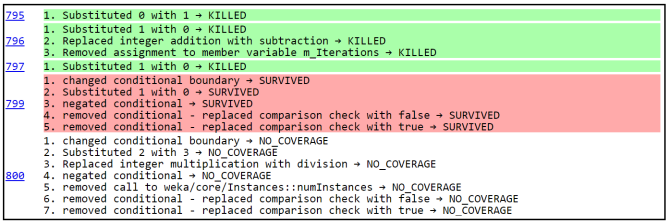


Fig. 6. Sample Mutation Testing Report

For recorded method-level tests without reduction shown in Table IV, we can see that most of the recorded tests for different methods and coverage criteria have a high mutant killing rate. Even without comparing to the original system-level execution, a small number of tests show high effectiveness in detecting potential faults that could occur in the selected methods. For four out of seven selected methods, recorded tests achieve over 80% of mutant killing rate for all the selected coverage criteria. The average mutant killing rate across seven methods are around 80% for all four different sets of tests that achieve edge coverage, edge-pair coverage, edge-set coverage, and these three combined. By only using edge coverage, the recorded method-level tests can achieve reasonably high mutant killing rate. With edge-pair and edge-set coverage, the mutant killing rate is further improved slightly in some cases. This indicates the method-level tests generated using our framework can effectively help developers to debug and find faults they are looking for, while significantly reducing the time and efforts required from developers for debugging.

For reduced method-level tests, their mutant killing rates are nearly the same as their original recorded tests. With differences no larger than 5% of their original killing rate. We even see some cases with increased mutant killing rate, such as for the edge-pair coverage of method "*cutPointsForSubset*". While coverage elements of our

TABLE IV. MUTATION TESTING RESULTS

Method	# of Mutants Generated for Covered Code	Statement Coverage	Edge Coverage Mutant Killing Rate		Edge-Pair Coverage Mutant Killing Rate		Edge-Set Coverage Mutant Killing Rate		Combined Recorded Tests Mutant Killing Rate	
			Recorded	*Reduced*	*Recorded*	*Reduced*	*Recorded*	*Reduced*	*Recorded*	*Reduced*
buildClusterer	269	69.70%	79.18%	79.18%	79.18%	79.18%	79.18%	79.18%	79.18%	79.18%
cutPointsForSubset	164	96.88%	81.71%	81.1%	81.71%	82.32%	85.98%	85.98%	85.98%	85.98%
EM_Init	102	88.68%	87.25%	87.25%	87.25%	87.25%	87.25%	87.25%	87.25%	87.25%
handleNumericAttribute	140	96.23%	89.29%	88.57%	90.71%	90.71%	91.43%	91.43%	91.43%	91.43%
select_working_set	128	96.15%	71.88%	75%	73.44%	75%	75%	78.91%	75%	78.91%
selectModel	132	86.21%	57.58%	57.58%	57.58%	57.58%	62.88%	62.88%	62.88%	62.88%
updateStatsForClassifier	122	69.70%	86.07%	81.15%	86.07%	84.43%	86.89%	86.07%	86.89%	86.89%
Average			79.00%	78.55%	79.42%	79.49%	81.23%	81.67%	81.23%	81.79%

specifically selected coverage criteria are maintained, other elements from other coverage criteria could become lost, or may be newly introduced after our binary reduction, such as combinations of the different branches being executed. The mutation testing results of the reduced tests show that even after the input sizes are significantly reduced, the coverage elements and also the fault detection effectiveness are still preserved. Our binary reduction technique on method-level tests can further help developers to reduce efforts for debugging while maintaining the debugging effectiveness of the method-level tests.

TABLE V. SYSTEM-LEVEL MUTATION TESTING

Method	Algorithm	# of Mutants Killed by System-Level Execution	# of Propagatable Mutants Killed by Combined Method-Level Tests
select_working_set	LibSVM	58	51
selectModel	J48	61	56

We also investigated the two methods *select_working_set* and *selectModel* with the lowest mutant killing rate by comparing their results to the mutation testing results of their system-level execution. We have planned on comparing all recorded method-level tests' mutation testing results with their corresponding system-level execution. However, while mutation testing is a very effective method to evaluate the quality of tests, mutation testing is a rather expensive method to use. In this paper, we only have two system-level mutation testing results for *select_working_set* and *selectModel*. Moreover, their system-level mutation tests both took over one week to complete. Note that some mutants that can be killed with method-level tests are not propagatable on the system level, i.e., a mutant may cause a method execution producing incorrect output, but such incorrect output on the method level did not cause an incorrect system-level output. We considered the option of recording all method executions of a method during its system-level execution. However, it is impractical, because of our selected methods have been executed with a large number of times, and many of them have large inputs as well. For comparing mutation testing results between method-level tests and system-level execution, we will only be considering the propagatable mutants for the method-level tests.

The system-level mutation testing results for *select_working_set* and *selectModel* are shown in Table V. For *LibSVM*, the system-level execution was able to kill 58 mutants, the combined method-level test of *select_working_set* was able to kill 51 out 58 propagatable mutants with a propagatable mutant killing rate of 87.93%. For *J48*, the system-level execution was able to kill 61 mutants, the combined method-level tests of *selectModel* were able to kill 56 out of 61 propagatable mutants with a propagatable mutant killing rate of 91.80%. The further investigation shows the reason why method-level tests recorded for *select_working_set* and *selectModel* have a lower mutant killing rate. It is likely because their original system-level execution has a lower mutant killing rate.

After investigating the un-killed propagatable mutants in the recorded method-level tests, we discovered three un-killed propagatable mutants from *select_working_set* and one from *selectModel* were mutations related to modifying boundary conditions. This means by adding more coverage criteria related to boundary conditions, a higher mutant killing rate can be achieved for the method-level test. With a few basic coverage criteria implemented for our framework, method-level tests produced by our framework can be very effective in detecting faults during debugging.

E. Performance Evaluation

We evaluate the performance of our implementation by investigating the original system-level execution time, the

time taken to evaluate and record the method-level tests, time taken to reduce tests, and the time taken to execute the recorded method-level tests. The results are shown in Table VI. Recall that in the experiments for mutation testing, both inputs and outputs of the selected method executions are recorded. However, the results shown in Table VI are only for recording the inputs and executing the recorded method-level tests with only inputs without comparing their outputs. This is because, in real-world use of our framework, outputs of the method executions do not need to be recorded.

As previously mentioned in Section II, we have two solutions for recording selected method executions. One approach is to serialize and temporarily store the inputs for each method execution and record the inputs locally when a method execution is determined to be significant. This method requires executing the entire system only once. However, in cases where a method has large inputs or is executed for a large number of times, this approach may have a significant performance issue due to all the unnecessary serialization. The other approach is to execute the entire system twice. In the first execution, we evaluate each method execution and store the execution IDs of the method executions. An execution ID is the index of a method execution based on the order of each method executions that happened during the system-level execution. In the second system-level execution, we only serialize and record the inputs of the selected method executions based on their execution IDs. The numbers marked with "*" indicates that the method-level tests were recorded using the second recording approach as shown in Table VI. The execution time is computed by subtracting the execution end time by the execution start time that was created using the Java *System.CurrentTimeMillis()* function.

TABLE VI. PERFORMANCE EVALUATION RESULTS

Method	Original Execution Time	Total Test Recording Time	Total Test Execution Time		Total Test Reduction Time
			Recorded	*Reduced*	
buildClusterer	5352 s	6303 s	5 s	5 s	27 s
cutPointsForSubset	9559 s	*21615 s	1836 s	5 s	7558 s
EM_Init	5356 s	5361 s	5 s	5 s	22 s
handleNumericAttribute	6357 s	*14624 s	155 s	2 s	1965 s
select_working_set	4491 s	*11531 s	176 s	1 s	2763 s
selectModel	6357 s	*14212 s	682 s	1 s	3122 s
updateStatsForClassifier	9559 s	*30513 s	875 s	3 s	4088 s

In Table VI, we see that recording method-level tests using our framework can take up to three times of the initial system execution. Additional test reduction time could take as much as two hours based on the size of the inputs (Our binary reduction utilizes serialization for deep copy as well). The reduced tests can be executed for many times during the debugging, the reduction time is a one-time investment, we believe the time is manageable for developers. Moreover, our approach is automated, allowing developers to work on other tasks while running our approach. For executing the recorded method-level tests, we see that it usually takes much less time than executing the entire system, especially for the reduced tests, the execution time can range from as little as one second to five seconds. Overall, we believe that recording and reducing method-level tests using our framework will help developers save a lot of time and efforts in debugging big data applications.

IV. RELATED WORK

We first review previous work related to generating tests for big data applications. Csallner et al. proposed an approach that uses dynamic symbolic execution to automatically generate tests for general *MapReduce* programs [1]. Morán et al. proposed *MRFlow,* a testing technique tailored to test *MapReduce* programs [5]. *MRFlow* uses data flow test criteria and oriented to transformations analysis between the input and the output in order to detect defects in *MapReduce* programs. Morán et al. also proposed a technique to generate different infrastructure configurations for a given *MapReduce* program that can be used to reveal functional faults [4]. They also proposed an automatic test framework that can detect functional faults automatically [3]. Chandrasekaran et al. proposed an approach to generate test input data using combinatorial testing for testing big data applications [6]. Previous work reported in [1, 2, 3, 4, 5] focuses on generating tests that help to identify functional faults, i.e., faults that will cause the program to generate unexpected outputs. In contrast, our work focuses on reducing debugging efforts for big data applications. Our tests are recorded in an effort to reproduce failures using a small number of method-level tests.

Second, some work has been reported on debugging big data applications. Gulzar et al. developed a tool, BigDebug, that simulates breakpoints to enable a developer to inspect a program without actually pausing the entire computation [7]. To help a user inspect millions of records passing through a data-parallel pipeline, BigDebug provides *guarded watchpoints*, which dynamically retrieve only those records that match a user-defined guard predicate. Chandrasekaran et al. proposed a technique that uses different annotators to debug the tracking data independently and their debugging results were collected for joint correction propagation for later analysis [9]. Our work is similar to Gulzar [7] and Li [9] in terms of only focusing on a subcomponent of the system. However, our work focuses on recording significant method-level executions to be replayed for debugging suspicious methods. Gulzar [7] and Li [9] focuses on tracking the changes made to certain objects using data flow analysis approach.

Third, our work is also related to existing work that records program information and uses the information to generate unit tests. Pasternak et al. proposed a technique that records interactions that take place during the execution of Java programs and uses these interactions to construct unit tests automatically using GenUTest [10]. Orso et al. proposed

a technique and conducted a feasibility study using SCARPE, a prototype tool, for selective capture and replay of program executions [6]. Similar to our work presented in this paper, Orso's technique [6] can be used to automatically generate unit tests based on the recorded information for testing purpose. Our work is similar to Pasternak [10] and Orso [6] in terms of recording method-level tests based on the system-level execution. However, our work focuses on recording unit tests for debugging one or more failures that have been observed instead of generating tests for triggering failures that have not been observed yet. Furthermore, our work also does not require complex instrumentation techniques on the target's bytecode [6]. Instead, we only employ simple instrumentation that keeps track of code coverage.

Finally, we review work related to reducing input size for the debugging purpose. Zeller et al. proposed Delta Debugging [8] technique to isolate failure-inducing inputs on the system level to reduce work required for debugging. Clause [14] et al. presented a technique based on dynamic tainting for automatically identifying subsets of a program's inputs that are relevant to a failure. These techniques reduce the debugging effort at the system level, in terms that the reduced datasets need to be executed at the system level. This is in contrast with our work that reduces the debugging effort at the method level.

V. Conclusion & Future Work

In this paper, we presented a framework to provide developers with method-level tests that were recorded from a failed system-level execution with the original dataset. These method-level tests preserve a given coverage criterion, e.g. edge, edge-pair, and edge-set coverage, and thus are likely to reproduce the failure observed at the system level. The binary reduction is used to further reduce method-level tests with large input. The set of method-level tests that are provided by our approach could help developers to effectively debug suspicious methods against properties of the original input dataset, and significantly reduce time and effort required for debugging big data applications.

There are two major directions for future work. First, we plan to conduct more experimental evaluation of our approach using more big data applications, datasets, and coverage criteria. Second, we plan to further automate our approach. In particular, we will develop techniques that can fully automate the instrumentation process. Our current approach still needs manual effort in modifying CFG generated by Atlas, inserting code for instrumentation, and identifying collection typed variables for reduction. It is our plan to make the tool publicly available.

VI. Acknowledgment

This work is supported by a research grant (70NANB15H199) from Information Technology Lab of National Institute of Standards and Technology (NIST).

Disclaimer: Certain software products are identified in this document. Such identification does not imply recommendation by the NIST, nor does it imply that the products identified are necessarily the best available for the purpose.

References

[1] Csallner, C., Fegaras, L., & Li, C. (2011, September). New ideas track: testing mapreduce-style programs. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering* (pp. 504-507). ACM.

[2] Jaganmohan Chandrasekaran, Huadong Feng, Yu Lei, Richard Kuhn, Raghu Kacker, "Applying combinatorial testing to data mining algorithms", *Software Testing Verification and Validation Workshops (ICSTW) 2017 IEEE Fourth International Conference on-6th International Workshop on Combinatorial Testing (IWCT),* 2017.

[3] Morán, J., Bertolino, A., de la Riva, C., & Tuya, J. (2017, July). Towards Ex Vivo Testing of MapReduce Applications. In *Software Quality, Reliability and Security (QRS), 2017 IEEE International Conference on* (pp. 73-80). IEEE.

[4] Morán, J., Rivas, B., De La Riva, C., Tuya, J., Caballero, I., & Serrano, M. (2016, August). Infrastructure-aware functional testing of mapreduce programs. In *Future Internet of Things and Cloud Workshops (FiCloudW), IEEE International Conference on* (pp. 171-176). IEEE.

[5] Morán, J., Riva, C. D. L., & Tuya, J. (2015, August). Testing data transformations in MapReduce programs. In *Proceedings of the 6th International Workshop on Automating Test Design, Selection and Evaluation* (pp. 20-25). ACM.

[6] Orso, A., & Kennedy, B. (2005, May). Selective capture and replay of program executions. In *ACM SIGSOFT Software Engineering Notes* (Vol. 30, No. 4, pp. 1-7). ACM.

[7] Muhammad Ali Gulzar, Matteo Interlandi, Seunghyun Yoo, Sai Deep Tetali, Tyson Condie, Todd Millstein, Miryung Kim. BigDebug: Debugging Primitives for Interactive Big Data Processing in Spark. Proceeding *ICSE '16 Proceedings of the 38th International Conference on Software Engineering*, Pages 784-795

[8] A. Zeller and R. Hildebrandt, "Simplifying and Isolating Failure-Inducing Input", IEEE Transactions on Software Engineering28(2), February 2002, pp. 183-200.

[9] Mingzhong Li, Zhaozheng Yin. Debugging Object Tracking by a Recommender System with Correction Propagation. In IEEE Transactions on Big Data (Volume: 3, Issue: 4, Dec. 1 2017)

[10] Pasternak, B., Tyszberowicz, S., & Yehudai, A. (2009). GenUTest: a unit test and mock aspect generation tool. *International journal on software tools for technology transfer*, *11*(4), 273.

[11] N. Li, U. Praphamontripong, and J. Offutt, "An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage," in Second International Conference on Software Testing Verification and Validation, ICST 2009, Denver, Colorado, USA, April 1-4, 2009, Workshops Proceedings, 2009, pp. 220–229.

[12] Eibe Frank, Mark A. Hall, and Ian H. Witten (2016). The WEKA Workbench. Online Appendix for "Data Mining: Practical Machine Learning Tools and Techniques", Morgan Kaufmann, Fourth Edition, 2016.

[13] Dua, D. and Karra Taniskidou, E. (2017). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.

[14] J. Clause and A. Orso. Penumbra: Automatically identifying failure relevant inputs using dynamic tainting. In ISSTA, pages 249–260, 2009.

[15] "Atlas Platform, EnSoft Corp." http://www.ensoftcorp.com.

[16] "PITest." http://pitest.org/.

[17] "FST, fast-serialization." https://github.com/RuedigerMoeller/fast-serialization.