BigTest: A Symbolic Execution Based Systematic Test Generation Tool for Apache Spark

Muhammad Ali Gulzar
University of California Los Angeles

Madanlal Musuvathi
Microsoft Research

Miryung Kim
University of California Los Angeles

ABSTRACT

Data-intensive scalable computing (DISC) systems such as Google's MapReduce, Apache Hadoop, and Apache Spark are prevalent in many production services. Despite their popularity, the quality of DISC applications suffers due to a lack of exhaustive and automated testing. Current practices of testing DISC applications are limited to using a small random sample of the entire input dataset which merely exposes any program faults. Unlike SQL queries, testing DISC applications has new challenges due to a composition of both dataflow and relational operators, and user-defined functions (UDF) that could be arbitrarily long and complex.

To address this problem, we demonstrate a new white-box testing framework called BigTest that takes an Apache Spark program as input and automatically generates synthetic, concrete data for effective and efficient testing. BigTest combines the symbolic execution of UDFs with the logical specifications of dataflow and relational operators to explore all paths in a DISC application. Our experiments show that BigTest is capable of generating test data that can reveal up to 2X more faults than the entire data set with 194X less testing time. We implement BigTest in a Java-based command line tool with a pre-compile binary jar. It exposes a configuration file in which a user can edit preferences, including the path of a target program, the upper bound of loop exploration, and a choice of theorem solver. The demonstration video of BigTest is available at `https://youtu.be/OeHhoKiDYso` and BigTest is available at `https://github.com/maligulzar/BigTest`.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; *Cloud computing*; • **Information systems → MapReduce-based systems**.

KEYWORDS

Test generation, symbolic execution, dataflow programs, data intensive scalable computing, map reduce

ACM Reference Format:
Muhammad Ali Gulzar, Madanlal Musuvathi, and Miryung Kim. 2020. BigTest: A Symbolic Execution Based Systematic Test Generation Tool for Apache Spark. In *42nd International Conference on Software Engineering Companion (ICSE '20 Companion), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3377812.3382145

ICSE '20 Companion, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7122-3/20/05.
https://doi.org/10.1145/3377812.3382145

1 INTRODUCTION

Data-intensive scalable computing (DISC) systems such as Mapreduce, Apache Hadoop [2], and Apache Spark [4] are popular today in large-scale big data applications. The current development process usually involves testing these applications on a small sample of data locally. Not surprisingly, such sampling is unlikely to yield adequate coverage, leading to errors in production. One way to address this problem is to build exhaustive testing tools. However, this is challenging for DISC applications. First, DISC applications consist of both dataflow operators such as `map`, `flatmap` and relational operators such as `join` and `group-by`. Second, unlike SQL queries where the use of user-defined functions (UDFs) is rare, DISC applications usually involve complex UDFs written in a general-purpose language such as C/C++, Java, or Scala which are susceptible to software bugs.

We demonstrate a new white-box testing technique for DISC applications called BigTest. BigTest first partitions a DISC program into a list of dataflow operators and a set of corresponding UDFs and performs symbolic execution on each extracted UDF in isolation. It then combines the path conditions of individual UDFs with the logical specifications of dataflow (or relational) operators such as `join`, `flatmap`, and `reduce`. Such combined exploration is more powerful than prior testing approaches [12, 13] that do not analyze the internal semantics of UDFs and consider them simply as uninterpreted functions.

The key contribution of BigTest comes from combining the internal semantics of user defined functions (UDFs) with the logical specification of dataflow and relational operators. Additionally, it models both the terminating and non-terminating cases for each dataflow operator. For instance in `join`, BigTest considers both non-terminating and terminating cases *i.e.,* keys matching in the left and the right table, and keys present in a one table only respectively. During symbolic execution, BigTest explicitly models unbounded collections created by `flatmap` and translates incremental aggregation logic in `reduce` into an iterative aggregator with a for-loop. Our combined exploration naturally results in a new notion of test coverage called Joint Dataflow and UDF (JDU) path coverage [11]. The final set of JDU path constraints is transformed into SMT queries and solved by an off-the-shelf theorem prover, Z3 [7] or CVC4 [6], to produce a set of concrete input records.

In our experiments, BigTest models 67% more JDU paths than the prior approach [12], thus revealing 2X more faults on average. We also show that only a few data records (order of tens) are actually required to achieve the same JDU coverage as the entire production data, highlighting BigTest's potential to minimize test data size by 10^5 to 10^8X. This is also reflected in CPU time savings of 194X on average, compared to testing code on the entire production

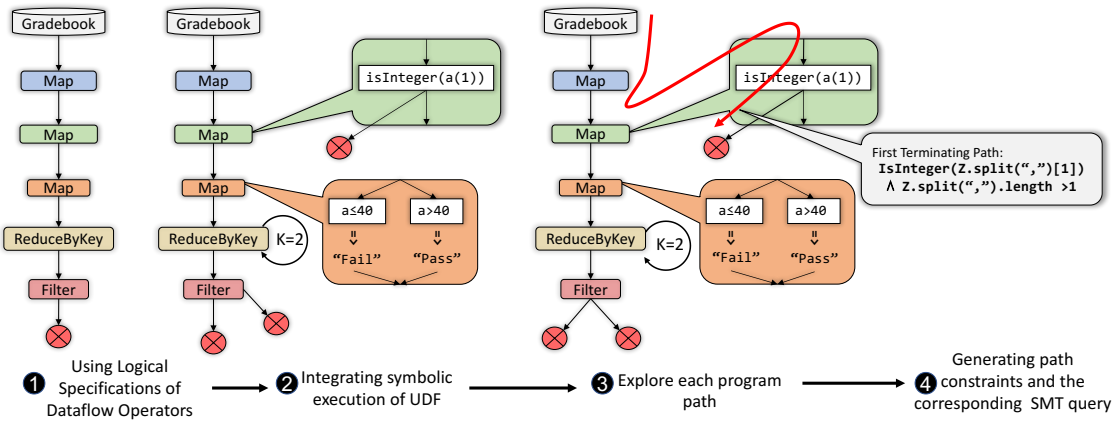

Figure 1: An overview of BigTest's approach on the motivating example. Red arrow represents the first terminating path with corresponding path conditions in a grey box.

data. Furthermore, BigTest synthesizes concrete input records in 19 seconds on average for all remaining untested paths, further increasing code coverage at interactive speed.

The full technical paper on this approach appeared at ESEC/FSE 2019 [11] and this paper describes BigTest's user interface and internal implementation with a focus on tool demonstration. BigTest is implemented in a Java-based command line tool. BigTest supports Scala Spark applications and can easily be extended to Java and other DISC frameworks such as Hadoop MapReduce [2] and Apache Flink [5]. The rest of the paper is organized as follows. Section 2 highlights BigTest's technical contributions. Section 3 describes implementation details. Section 4 demonstrates a usage scenario. Section 5 describes evaluation settings and results. Section 6 concludes the paper.

2 TECHNICAL APPROACH

We briefly describe BigTest's main contributions below. The detailed explanation of its approach is described in our full paper [11]. BigTest takes in an Apache Spark program written in Scala as an input, generates a set of path constraints up to a given bound, and constructs concrete test data using an off-the-shelf theorem prover. Figure 1 illustrates BigTest in four steps.

2.1 Program Decomposition

A dataflow program is comprised of dataflow operators such as `map`, `group-by`, and `reduce` and corresponding user defined functions. These dataflow operators are implemented by over 700K lines of code in Apache Spark which includes support for job scheduling, fault tolerance, and data partitioning. Due to this complex system-stack, the symbolic execution of these dataflow operators is infeasible. Instead, BigTest abstracts Apache Spark framework's code in terms of clean logical specifications. We decompose a DISC application into a dataflow skeleton and combine the symbolic execution of individual UDFs with the logical specifications. This process is illustrated in ❶ and ❷ of Figure 1.

As a first step, BigTest performs Abstract Syntax Tree (AST) analysis to extract each UDF into a separate Java class and performs symbolic execution using Symbolic PathFinder (SPF) [14]. BigTest also generates a configuration file required by SPF for symbolic execution and performs dependency analysis to include external classes and methods referenced in the UDF. For aggregation operators, we translate a corresponding UDF into an iterative representation with a loop. For example, the UDF for `reduce` is an associative binary function that is turned into a for-loop.

2.2 Path Constraint Generation

For each extracted UDF, we perform symbolic execution using its Java class and the corresponding JPF file to construct a set of path constraints and effects. BigTest captures path conditions and effects from SPF through a custom listener. It plugs in the path conditions and effects of a UDF into the logical specifications of an operator to produce the complete path conditions for that operator, as seen in ❶ and ❷ of Figure 1. The constraints of an operator represent equivalence classes derived from the logical specifications of that operator. These logical specifications reflect the semantics of dataflow (and relational) operators and how they interact with UDFs [11]. Finally, BigTest combines the path conditions of each operator with the incoming constraints from its upstream operator. Figure 1 shows the constraints from one such path.

2.3 Test Data Generation

BigTest rewrites each end-to-end path constraint into an SMT query using relevant arithmetic and logical operators available in SMT. For unsupported operations such as string `split` and `isInteger`, BigTest introduces a library of SMT functions. It models individual elements of an array up to a user defined bound K. BigTest executes each SMT query separately and finds satisfying assignments (*i.e.,* test inputs) to exercise a particular path.

3 IMPLEMENTATION

BigTest is a JVM-based command line tool that supports the symbolic execution of dataflow programs. We extend Symbolic PathFinder (SPF) [14] with a symbolic dataset class containing methods for each dataflow operators such as `map`, `reduce`, and `join`. Each method takes a symbolic representation of the corresponding UDF as the only argument and programmatically integrates it with the logical specification of a dataflow operator. The return type of these

```
1  sc.textFile("hdfs://registrar:9999/gradebook.log")
2      .map {  line => val arr = line.split(",")
3                      arr(1)
4      }
5      .map {     l => val a = l.split(":")
6                   (a(0), Integer.parseInt(a(1)))
7      }
8      .map {     a => if (a._2 > 40)
9                      ("Pass".concat(a._1), 1)
10                  else
11                     ("Fail".concat(a._1), 1)
12     }
13     .reduceByKey(_+_)
14     .filter (  v => v._2 <=2 && v._1.startsWith("Fail") )
```

Figure 2: A Spark program that identifies the courses with less than two failing students.

methods is a symbolic dataset which contains the complete symbolic representation including path constraints and effects until that point of the program. This implementation is inspired by Apache Spark's resilient distributed dataset (RDD) [16]. BigTest requires a `conf` file that includes a loop iteration bound and sample inputs to each UDF. A user can use the pre-compiled jar file to invoke BigTest.

```
java -jar BigTest.jar -enableBT <Program Directory>
```

First, BigTest reads the bytecode of the given program and translates it to Java source code with Java Decompiler (JAD) [3]. Using Eclipse Java Development Toolkit (JDT) [1], it parses the abstract syntax tree of Java source code and extracts corresponding UDFs, creating a collection of Java classes. BigTest invokes extended SPF on each UDF class to explore all paths in the UDF symbolically and combines them with the logical specifications of dataflow operators. It then constructs end-to-end path constraints of the entire program in SMT2 Lib format and uses SMT solvers (CVC4 and Z3) to find satisfying assignments. BigTest can easily be attached to other theorem solvers as it stores path constraints in multiple SMT2 files under `\tmp\` or a specific directory name defined by a user in the `conf` file.

4 DEMONSTRATION

In this section, we present a step-by-step demonstration of BigTest. Suppose Alice writes a DISC application in Apache Spark to find courses with fewer than two failing students. She uses the entire university gradebook database which contains several years of grading information spanning gigabytes. A sample row in this dataset contains comma-separated fields of a course id, a final mark in percentiles, year, a student id, a session name, and a major. Below is a small sample of the gradebook data.

```
CS233:77,1994,80554313,F1994,CS,..
CS233:53,1994,80594911,F1994,EE,..
CS233:29,1994,30472981,F1994,BIO,.
```

To perform this analysis, Alice writes a program, as shown in Figure 2. First, she loads the data from an HDFS storage using Apache Spark's `textFile` API in line 1. Once the data is loaded in Spark, she uses a `map` operator to extract the course id and a student's mark from each row using a UDF (lines 2 to 4). In the following `map` operation (lines 5 to 7), a string such as "CS233:77" is transformed into a tuple of a string and an integer. In lines 8 to 11, Alice annotates the tuples with either a "Pass" or a "Fail" string

```
1  filter1 = "",1
2  map3 = "",1
3  map4 = "CS:123"
4  reduceByKey2 = {1,2,3,4}
5  map5 = "a,a"
6  K_BOUND = 2
```

Figure 3: A configuration file for the motivating example

based on the marks. For each course, she calculates the total sum of passing and failing students using `reduceByKey` and then applies `filter` to find the courses with fewer than two failing students in line 14. This program has a total of 17 JDU paths, where 2 are non-terminating and 15 are terminating paths.

Currently, Alice can test her program on the entire dataset which may take a few hours to compute. However, in case of a test failure, it is nearly impossible for her to identify the failing input record as the input data contains several million records. Such technique is neither efficient nor effective. As an alternate approach, she tests her program using top 100 records from the input data. The program produces an empty result without any failure since the test data comprises passing students only. As a result, only 12 out of 17 JDU paths are exercised, consequently missing three crash-inducing cases emerging from statements such as `Interger.parseInt`. It is critically important to cover such paths because, at large scale, a crash-inducing record may crash the entire job and throw away several hours of progress.

To run BigTest on her program, Alice first writes a configuration file in the format shown in Figure 3. She writes `K_BOUND=2` to limit the symbolic exploration of unbounded loops and collections to avoid path explosion. SPF requires sample input arguments to a function before symbolically executing it. BigTest takes such input arguments from Alice through `conf` file and passes them to SPF. A UDF can be identified with its corresponding operator name followed by the execution order number in reverse (similar to Spark's execution). Figure 3 shows the final configuration file. Alice invokes BigTest's command line tool using the following command.

```
java -jar BigTest.jar -enableBT /GradeAnalysis
```

Console Log 1

```
Map4
Non-terminating:
PC: {l splitn 1 : isinteger && l = x2}
E: {x3_1 = l splitn 0 :, x3_2 = l splitn 1 : str.to.int }
Terminating:
PC: {l splitn 1 : notinteger} E: {}
PC: {l equals ""} E: {}
Map3
Non-terminating:
PC: {a_t2 <= 40 && a_t1 = x4_1 && a_t2 = x4_2}
E: {x5_1 = Fail str.++ a_t1, x5_2 = 1}
PC: {a_t2 > 40 && a_t1 = x4_1 && a_t2 = x4_2}
E: x5_1 = Pass str.++ a_t1, x5_2 = 1
```

Alice can monitor the progress of BigTest by looking at the console output log. BigTest logs progress with the number of paths explored, the path constraints, and the effects at each operator level until the final operator is reached. Console Log 1 shows the explored paths from the second and third map operator. `PC` refers to a path condition and `E` represents corresponding effect. For example, path constraint `l splitn 1 : notinteger` represents a terminating path emerging from `Integer.parseInt` at line 6 of Figure 2. Near the end of the execution, Alice starts to see the entire program's

path constraints built on top of the individual path conditions of underlying operators. A sample console log snippet of a final non-terminating path constraint is shown below.

Console Log 2

```
PC: {l1 splitn 1 : isinteger && a2l1 <= 40 && l2 splitn 1 :
isinteger && a2l1 <= 40 && a select 1 = arr[1] 1 <
arr_length && a select 0 = arr[0] && x7 < 2 && x8
str.substr 0 4 equals Fail }
E: {x7 = arr[1] + arr[0] && lp1 = l1 splitn 0 && a2l1 = lp1
splitn 1 : str.to.int && ...}
```

For every final path of the program, BigTest calls an SMT solver to produce a set of concrete input rows that Alice can use as a test data for her unit test.

Console Log 3

```
Path : 13
running CVC4 $>cvc4 --strings-exp --lang smt2 < /tmp/-966362206
line_1 () String ":41"
line_2 () String ":41"
line_3 () String ":0"
line_4 () String ":0"
```

Alice copies the test data generated by BigTest for path 13 (see Console Log 3) into a test data file and uses that file as input to a unit test. Due to small size, Alice comfortably runs this test on her local machine. Although the input does not contain any course with less than two failing students, the test output includes one row instead of zero. Upon further investigation, Alice identifies a code fault where she mistakenly used ≤ instead of < (line 14 in Figure 2). Similarly, BigTest generates test inputs such as empty string or non-numeric string to reveal critical corner cases which can lead to a program crash at lines 3 and 6 in Figure 2. Alice fixes such cases through relevant exception handling and data filtering to eliminate the possibility of costly runtime crashes.

5 RELATED WORK

BigTest is inspired by Li et al.'s representation of dataflow operators in their work Sedge [12]. They consider UDFs as black box and encodes them into *uninterpreted functions*. By treating UDFs as black-box, Sedge overlooks many code faults present in the UDFs as UDFs are more prone to human error. Our full-scale evaluation [11] shows that Sedge is unable to cover 78% of JDU paths revealing only 50% of the faults compared to BigTest. Furthermore, it also lacks support for commonly used aggregation operators such as `flatmap`, `reduce`, and `reducebyKey`. Olston et al. suggested a similar approach as Sedge and suffers from the same challenges [13].

Using dataflow queries in traditional software is not uncommon; however, such queries rarely contain user-defined functions. Emmi et al. perform concolic execution of a program embedded with an SQL query by symbolically executing the program and using a set of pre-defined rules for SQL queries [8]. Their approach is only applicable to basic SQL operations e.g., SELECT...FROM...WHERE. Several previous test generation approaches use symbolic execution on traditional software [15]. However, as mentioned before, an off-the-shelf symbolic execution tool is not suited for DISC applications because it would symbolically execute the entire codebase of an underlying DISC framework. Our prior work focused on automated and interactive debugging [9, 10], while this work improves the testing of DISC applications.

6 CONCLUSION

Efficient and effective testing of big data analytics is still in the early stage of development. We demonstrate a novel tool for white-box testing of big data analytics. BigTest takes an Apache Spark application as an input and systematically explores the combined behavior of both dataflow operators and the corresponding UDFs to generate path constraints. It leverages off-the-shelf theorem solvers to generate concrete test inputs from path constraints. In our experiments, test data generated by BigTest reveals 2X more faults than prior approaches while consuming 194X less CPU time, on average, compared to testing on the entire dataset.

Acknowledgments. We thank the anonymous reviewers for their comments. The participants of this research are in part supported by Google PhD Fellowship, NSF grants CCF-1764077, CCF-1527923, CCF-1460325, CCF-1723773, ONR grant N00014-18-1-2037, Intel CAPA grant, and Samsung grant.

PUBLICATIONS

[1] 2019. Eclipse Java development tools (JDT). https://www.eclipse.org/jdt/.
[2] 2019. Hadoop. http://hadoop.apache.org/.
[3] 2019. Java Decompiler. http://java-decompiler.github.io/.
[4] 2019. Spark. https://spark.apache.org/.
[5] 2020. Apache Flink. https://flink.apache.org/.
[6] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovi'c, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*, Vol. 6806. Springer, 171–177. Snowbird, Utah.
[7] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.
[8] Michael Emmi, Rupak Majumdar, and Koushik Sen. 2007. Dynamic Test Input Generation for Database Applications. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis* (London, United Kingdom) *(ISSTA '07)*. ACM, New York, NY, USA, 151–162. https://doi.org/10.1145/1273463.1273484
[9] Muhammad Ali Gulzar, Matteo Interlandi, Xueyuan Han, Mingda Li, Tyson Condie, and Miryung Kim. 2017. Automated Debugging in Data-intensive Scalable Computing. In *Proceedings of the 2017 Symposium on Cloud Computing* (Santa Clara, California) *(SoCC '17)*. ACM, New York, NY, USA, 520–534.
[10] Muhammad Ali Gulzar, Matteo Interlandi, Seunghyun Yoo, Sai Deep Tetali, Tyson Condie, Todd Millstein, and Miryung Kim. 2016. BigDebug: Debugging Primitives for Interactive Big Data Processing in Spark. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. ACM, New York, NY, USA, 784–795. https://doi.org/10.1145/2884781.2884813
[11] Muhammad Ali Gulzar, Shaghayegh Mardani, Madanlal Musuvathi, and Miryung Kim. 2019. White-box Testing of Big Data Analytics with Complex User-defined Functions. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. ACM, New York, NY, USA, 290–301. https://doi.org/10.1145/3338906.3338953
[12] Kaituo Li, Christoph Reichenbach, Yannis Smaragdakis, Yanlei Diao, and Christoph Csallner. 2013. SEDGE: Symbolic example data generation for dataflow programs. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. IEEE, 235–245.
[13] Christopher Olston, Shubham Chopra, and Utkarsh Srivastava. 2009. Generating Example Data for Dataflow Programs. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data* (Providence, Rhode Island, USA) *(SIGMOD '09)*. ACM, New York, NY, USA, 245–256.
[14] Corina S. Păsăreanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person, and Mark Pape. 2008. Combining Unit-level Symbolic Execution and System-level Concrete Execution for Testing Nasa Software. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA '08)*. ACM, New York, NY, USA, 15–26.
[15] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. 2004. Test Input Generation with Java PathFinder. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis* (Boston, Massachusetts, USA) *(ISSTA '04)*. ACM, New York, NY, USA, 97–107. https://doi.org/10.1145/1007512.1007526
[16] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (San Jose, CA) *(NSDI'12)*. Berkeley, CA, USA, 2–2.