# Symbolic Execution of Apache Spark Programs

**Omar A. Erminy Ugueto**
March 5, 2017

**Fachbereich Informatik**

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Omar Erminy
Matriculation Number: 2996125
Study Program: Master in Distributed Software Systems

Master Thesis
Topic: Symbolic Execution of Apache Spark Programs

Submitted: March 5, 2017

Supervisor: Prof. Dr. Guido Salvaneschi

Prof. Dr-Ing. Mira Mezini
Fachgebiet Softwaretechnik
Fachbereich Informatik
Technische Universität Darmstadt
Hochschulstraße 10
64289 Darmstadt

## Abstract

Informationen zu Inhalten der Zusammenfassung entnehmen Sie bitte Kapitel 6.1 des Skripts zur Veranstaltung *Wissenschaftliches Arbeiten und Schreiben für Maschinenbau-Studierende*.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

# Contents

# 1 Introduction

# 2 Related Work

## 2.1 Apache Spark

Spark is a data processing framework that was first introduced in 2012 [1]. Similar to other systems, such as MapReduce [2] and Dryad [3], it aims to provide a clean and flexible abstraction to distributed computations on large datasets. However, Spark offers two advantages in comparison to such systems: It makes use of a shared memory abstraction that improves performance by avoiding persisting intermediate sets. It also maintains an efficient fault-tolerance mechanism, based on tracking coarse-grained operations, that can recover lost tasks with minimal impact.

The working units in Spark are called *Resilient Distributed Datasets*, better known as RDDs. These units represent an immutable partitioned collection of elements in a distributed memory space. RDDs can only be created through a set of deterministic operations, known as *transformations* (e.g., *map*, *filter* and *join*), that can be applied to both, raw data or other RDDs. Transformations are not evaluated immediately, instead Spark keeps track of all the transformations applied to each RDD in a program so it can optimize their subsequent processing. Additionally, RDDs can be made persistent into storage or can be operated to produce a value. This kind of operations are known as *actions* (e.g., *count*, *reduce* and *save*), and they are the ones that trigger the processing of RDDs.

To interact with the RDD abstraction, Spark provides several APIs for different programming languages such as Java, Scala, Python and recently R [4]. Listing 2.1 presents a simple Spark program written with the Scala API, that processes log files in the search for errors. The operation in line 1 creates the first RDD from a log file, whose origin could be a local file or a partitioned file in a distributed file system such a Hadoop Distributed File System (HDFS) [5]. Spark converts each line in the file to a *String* element in the newly created RDD. In lines 2 to 4, a chain of transformations is applied to the RDD: First, elements not containing the text “ERROR” are filtered. Next, the remaining elements are transformed to tuples consisting of a certain property (e.g., error type; assumed to be the first information in a log entry) and the number 1. Finally, the tuples are grouped and counted based on the chosen property. Line 5 represents the action applied to the RDD, in this case, saving it to persistent storage.

```
1  val log = spark.textFile("*file*")
2  val errors = log.filter(_.contains("ERROR"))
3    .map(error => (error.split('\t')(0),1))
4    .reduceByKey(_+_)
5  errors.save()
```

**Listing 2.1:** Entries in a log file are filtered, grouped and counted based on a common property. Finally the result is saved to persistent storage.

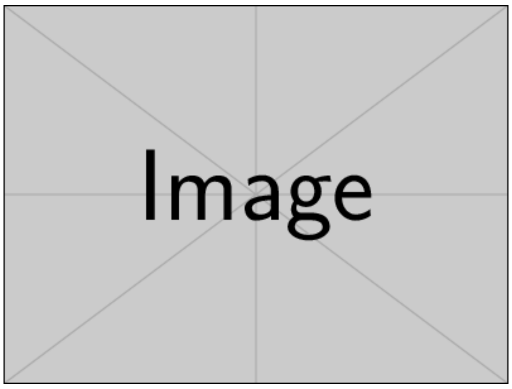

**Figure 2.1:** Lineage of a simple Spark program.

During the execution of a program, Spark does not generate imperatively new data collections for every transformation it finds. Instead, it constructs new RDDs attached with the operation that has to be applied to each element. The resulting RDD is a sequence of operations starting from the source dataset, whose semantics depends on the nature of each transformation involved. It is not until an action is found that the target RDD is resolved and the whole sequence of transformations actually operates the data.

Delaying the resolution of RDDs in this way allows Spark to improve the distribution of operations in a clustered dataset, taking advantage of properties like data locality. Moreover, the trace of operations that produced a certain element in an RDD, known as *lineage*, enables Spark to recover failed tasks only recalling to the necessary data elements that reproduce the lost portion. Figure 2.1 depicts the resulting lineage of the program explained in listing 2.1.

Most of the operations in Spark are higher-order functions, this means they accept one or more functions as parameters. For example, the *filter* transformation requires a function that takes an element of the RDD and evaluates to a boolean value. These user-defined functions work as closures by scoping their environment even if it contains references to variables outside itself; this enables Spark to ensure consistency when applying such functions in parallel nodes. The use of higher-order functions serve as a flexible mechanism to adapt Spark's computation model to different tasks.

The inherent capacity of Spark to operate in a distributed memory space makes it well-suited for two particular scenarios: iterative algorithms and interactive querying. The former, which are commonplace among machine learning algorithms, leverages on the reuse of datasets and avoids having to perform costly I/O operations for every iteration. The latter, allows data mining techniques to synthesize queries faster by keeping working data at hand.

Spark is part of the Apache Software Foundation and it is offered as an open-source software [6, 7]. Several purpose-specific libraries are built on top of Spark, as is the case of: MLlib for machine learning [8], GraphX for graph computations [9], Spark Streaming for stream processing [10], and Spark SQL, an SQL-like interface for structured querying in Spark [11].

In 2014, Spark reported the fastest Daytona GraySort as defined by the Sort Benchmark committee, and later in 2016, Spark was part of the technology stack that claimed the most resource-efficient Daytona CloudSort as defined by the same committee [12, 13, 14]. Overall, Spark offers a better performance in comparison to other data processing frameworks.

## 2.2 Formal Methods - Symbolic Execution

(Coming from Finite Models. [16])

(Model Checking, why should we use it?)

(Model) A model is a representation that is simple than the artifact it represents but preserves (or at least approximates) some important attributes of the actual artifact. [16]

(Models summary) Models play many of the same roles in software development as in engineering of other kinds of artifacts. Models must be much simpler than the artifacts they describe, but must preserve enough essential detail to be useful in making choices. For models of software execution, this means that a model must be abstract away enough detail to represent the potentially infinite set of program executions states by a finite and suitably compact set of model classes. ...Some models, like CFG, can be extracted from programs. Tne key trade-off for these extracted models is precision versus the cost of producing and storing the model. [16]

(Related to Model Checking — First of all, programs often contain fatal errors despite the existence of careful designs.Many deadlocks and critical section violations, for example, are introduced at a level of detail which designs typically do not deal with, if formal designs are made at all.) [15]

(Related to Model Checking — The other kind of error is more simple minded concurrency programming errors, such as forgetting to put code in a critical section or causing deadlocks. Errors of this kind will typically not be caught in a design, and they are a real hazard, in particular in safety critical systems.) [15]

(Related to finite abstractions in model checking and the state space) a single program execution can be viewed as a sequence of states alternating with actions (e.g., machine operations). The possible behavior of a program are a set of such sequences. If we abstract from the physical limits of a particular machine, for all but the most trivial programs the set of possible execution sequences is infinite. That whole set of states and transitions is called the state space of the program. Models of program executions are just abstractions of that space. [16]

(State and transitions) A transition from one state node “a” to another state node “b” denotes the possibility that a concrete program state corresponding to “a” canbe followed immediately by a concrete program state corresponding to “b”. Usually we label the edge to indicate a program operation, condition, or event associated with the transition. We may label transitions with both an external event or a condition (what must happen or be true for the program to make a corresponding state change) and with a program operation that canbe thought of as a “response” to the event. Such finite state machine with *event/response* labels on transitions is called a Mealy machine. [16]

(About properties of model themselves, how correct they are) There are three kinds of correctness relations that we may reason about with respect to finite state machine models. The first is the internal properties, such as completeness and determinism. Second, the possible executions of a model, described by paths through the FSM, may satisfy (or not) some desired property. Third, the finite state machine model should accurately represent possible behaviors of the program. [16]

(Finite State Verification) Finite State Verification borrows techniques from symbolic execution and formal verification, but like control and data flow analysis, applies them to models that abstract the potentially infinite state space of a program behavior into finite representations. [16]. Systematically exploring an enormous space of possible programs states, the challenge is to construct a suitable model of software that can be analyzed with reasonable expenditure of human and computational resources, captures enough significant detail for verification to succeed, and can be shown to be consistent with actual software.

(State Space Exploration — Benefit) A few seconds of automated analysis to find critical faults that can elude extensive testing seems a very attractive option.

(State Space Explosion) hmmmm

(Program Analysis) Conventional program testing is weak at detecting program faults that cause failures only rarely or only under conditions that are difficult to control. For example, conventional programming testing is not an effective way to find race conditions between concurrent threads that interfere only in small critical sections, or to detect memory access faults that only occasionally corrupt critical data structures. These faults lead to failures that are sparsely scattered in a large space of possible program behaviors, and are difficult to detect by sampling, but can be detected by program analyses that fold the enormous program state space down to a more manageable representation. [16]

—————-

(Symbolic Execution) (Definition) Symbolic execution builds predicates that characterize the conditions under which execution paths can be taken adn the effect of the execution on program state. Extracting predicates through symbolic execution is the essential bridge from complexity of program behavior to the simpler and more orderly world of logic. It finds important applications in program analysis, in generating test data, and in formal verification (proof) of program correctness. [16].

(Another definition) Symbolic execution is a bridge from an operational view of program execution to logical and mathematical statements. [16].

(Usefulness) Conditions under which a particular control flow path is taken can be determined through symbolic execution. This is useful for identifying infeasible program paths (those that can never be taken) and paths that could be taken when they should not. It is fundamental to generating test data to execute particular parts and paths in a program. [16].

(Use cases — not so interesting) Symbolic execution is a fundamental technique that finds many different applications. Test data generators use symbolic execution to derive constraint on input data. Formal verification systems combine symbolic execution to derive logical predicates with theorem provers to prove them. Many development tools use symbolic execution techniques to perform or check program transformations, for example, unrolling a loop for performance or refactoring source code. [16].

(More cases) Although full verification is unfeasible or even useful. “Nonetheless the basic methods of formal verification, including symbolic execution, underpin practical techniques in software analysis and testing. They find use in several domains:

- Rigorous proof of properties of (small) critical subsystems, such as a safety kernel of a medical device.

- Formal verification of critical properties (e.g., security properties) that are particularly resistant to dynamic testing.

- Formal verification of algorithm descriptions and logical designs that are much less complex than their implementations in program code.

. [16].

(The basis of symbolic execution relies in tracing execution with symbolic values and expressions). [16]. When tracing execution with concrete values, it is clear enough what to do with a branch statement, for example, an if or while test: The test predicate is evaluated with the current values, and the appropriate branch is taken. If the values bound to variables are symbolic expressions, however, both the True and False outcomes of the decision may be possible. Execution can be traced through the branch in either direction, and execution of the test is interpreted as adding a constraint to the record outcome. [16].

(Show example)

(Satisfying the predicate) One can think of “satisfying” the predicate by finding concrete values for the symbolic variables that make it evaluate to True; this corresponds to finding data values that would force execution of a program path. If no such satisfying values are possible, then that execution path cannot be executed with any data values; we say it is an **infeasible** path. [16].

## 2.3 Java PathFinder

Developed at JPF acr:nasa’s Ames Research Center [17], Java PathFinder (JPF) is an execution environment for verification and analysis of Java bytecode programs [15, 18]. Since its publication in the year 2000 [19], JPF has evolved from being a model translator to a fully fledged, highly customizable virtual machine capable of controlling and augmenting the execution of a program.

Java is a widely known, general-purpose programming language with strong roots on concurrency support and object-oriented principles [20]. Programs written in Java are compiled to the standardized instruction set of the Java Virtual Machine (JVM), known as Java bytecode. This process makes Java programs portable between architectures implementing the JPF acr:jvm specification. A JPF acr:jvm implementation serves as an interpreter of Java bytecode and allows the optimization and execution of the program tailored for the host platform [21].

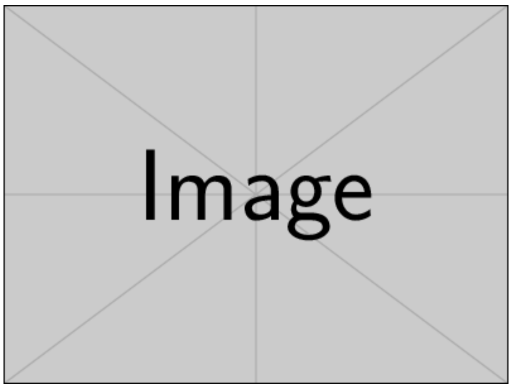

**Figure 2.2:** JPF Workflow

JPF focuses on Java mainly for three reasons: its wide adoption as a modern programming language, its simplicity in comparison to other high profile languages, and the flexibility in terms of bytecode analysis; potentially enabling the verification of any other language capable of being compiled into Java bytecode. Moreover, the non-trivial nature of concurrent programs makes them difficult to construct and debug. A model checker with the capacity of validating concurrent Java programs is crucial for ensuring correctness of mission-critical software, such as the likes required by JPF acr:nasa.

In its core, JPF is a Java Virtual Machine implemented in Java itself, comprised of several extensible components that dictate the verification strategy to be followed. The fact that JPF is written in Java means that it is executed on a canonical JPF acr:jvm; in other words, a JPF acr:jvm on top of a JPF acr:jvm.

The default mode of operation of JPF is *explicit state model checking*. This means that JPF keeps track of the execution status of a program, commonly referred to as a state, to check for violations of predefined properties. A state is characterized by three aspects: the information of existing threads, the contents of the heap, and the sequence of previous states that led to the current execution point (also known as path). A change in any of the aforementioned aspects represents a transition to a new state. Additionally, JPF associates complementary information to a state (e.g., range of possible values that trigger transitions), in order to reduce the total number of states to be explored. Termination is ensured by avoiding revisiting states.

Figure 2.2 portrays the elements that participate in a verification process using JPF . The program under test is loaded into JPF 's core, where its instructions are executed one by one until an execution choice is found. At this point, JPF records the current state and attempts to resume execution, exploring all possible scenarios based on the choice criteria. Once a chosen path has been completely explored, JPF backtracks to a recorded state, in order to explore a new path.

Listing 2.2 introduces an example that illustrates better how JPF acr:jpf works. The program analyzed represents a trivial division of two random values. However, the problem relies on the fact that, under some specific values, the operation could yield invalid. Problems like this, where computations depend on random and unbounded values, are common sources of bugs in real software and, in many cases, are difficult to identify. With the right configuration, JPF acr:jpf could detect this kind of problems by exploring the range of possible values that a random integer could take. Lines 6 and 7 indicate that random values have been generated; at this point JPF acr:jpf could start exploring all different possible

```java
import java.util.Random;

public class RandomExample {
  public static void main(String[] args) {
    Random random = new Random();
    int a = random.nextInt(2);
    int b = random.nextInt(3);
    int c = a/(b+a-2);
  }
}
```

**Listing 2.2:** The use of random values could lead to unexpected behavior. In this case, a division by zero could occur if certain combinations of random values are used.(Example taken from JPF)

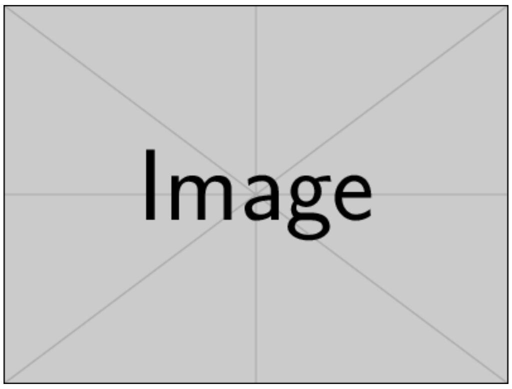

**Figure 2.3:** State space of the random example

combinations spanning the range of all integer values, but clearly this would imply an enormous number of combinations that would result in a state space explosion. To avoid this, a choice generator is registered, using the value 0 and parameter passed to the *nextInt* function as bounds for the set of integer random values that can be obtained. Consequently, a combination that triggers the invalid operation is found promptly and reported back to the user. Figure 2.3 depicts the corresponding state graph that would result from validating the program with JPF .

A key aspect of JPF was to make it extensible and customizable. With a modular design, users of the tool are capable of tuning JPF up to the needs of a wide variety of analyses and verifications. The main extension points are:

- **Bytecode Factories**: Define the semantics of the instructions executed by JPF 's virtual machine. Modifications to the bytecode factory define de execution model of the analyzed program (e.g., operations on symbolic values).

- **Choice Generators**: A set of possible choices must be provided in order to explore different behaviors of the system under test (e.g., a range of integer values for validation of random input). This aspect is critical reduce the number of states explored during a validation, hence scoping the reach of an analysis.

- **Listeners**: Serve as monitoring points for interacting with the execution of JPF . Listeners react to particular events triggered during the execution of an analysis, providing the right environment for the assertion of different properties.

- **Native Peers**: In some cases, a system under test will contain calls that are irrelevant to the analysis carried out (e.g., calling external libraries) or will execute native instructions that cannot be interpreted by JPF . For these cases, native peers provide a mechanism for modeling the behavior of such situations and efficiently delegating their execution to the host virtual machine.

- **Publishers**: Report the outcome of an analysis. Whether a property was violated or the system under test was explored satisfactorily, publishers provide the information that makes the analysis valuable.

- **Search Strategies**: Indicate how the state space of the system under test is to be explored. In other words, the search strategy tells JPF when to move forward and generate a new state or when to backtrack to a previously known state in order to try a different choice. Search strategies can be customized to guide the exploration of the state space to areas of interests where the analysis is most likely to detect an anomaly.

Although *explicit state model checking* is JPF 's default mode of operation, by no means is the only one. Different kinds of formal methods can used or implemented through modules, which are sensible extensions to JPF 's core that accomplish a particular task. The modules range from different execution models to the validation of specific properties not included previously in the core. Some examples are: JPF-Racefinder, an extension for precisely detecting data races, and Symbolic PathFinder (SPF), which gives support to the *symbolic state model checking* operation mode. The latter of these examples is explained further in the next section.

### 2.3.1 Symbolic PathFinder

(Definition of SPF)

(Explain its extension points: Choice Generators, Listeners, Symbolic Instruction Factory)

(Mention the solvers)

# 3 Evaluation

## 4 Future Work

# Bibliography

[1] Zaharia, M. et al. “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing”. In: *NSDI’12 Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation* (2012), pp. 2–2. ISSN: 00221112. DOI: `10.1111/j.1095-8649.2005.00662.x`. arXiv: `EECS-2011-82`.

[2] Dean, J. and Ghemawat, S. “MapReduce: Simplied Data Processing on Large Clusters”. In: *Proceedings of 6th Symposium on Operating Systems Design and Implementation* (2004), pp. 137–149. ISSN: 00010782. DOI: `10.1145/1327452.1327492`. arXiv: `10.1.1.163.5292`.

[3] Isard, M. et al. “Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks”. In: *ACM SIGOPS Operating Systems Review* (2007), pp. 59–72. ISSN: 01635980. DOI: `10.1145/1272998.1273005`.

[4] Venkataraman, S. et al. “SparkR: Scaling R Programs with Spark”. In: *Sigmod* (2016), p. 4. ISSN: 07308078. DOI: `10.1145/1235`. arXiv: `arXiv:1508.06655v1`.

[5] *Welcome to Apache™ Hadoop®!* URL: `http://hadoop.apache.org/` (visited on 2017).

[6] *Welcome to The Apache Software Foundation!* URL: `https://www.apache.org/` (visited on 2017).

[7] *Apache Spark™ - Lightning-Fast Cluster Computing*. URL: `http://spark.apache.org/` (visited on 2017).

[8] Meng, X. et al. “MLlib : Machine Learning in Apache Spark”. In: *Journal of Machine Learning Research* 17 (2016), pp. 1–7. arXiv: `arXiv:1505.06807v1`.

[9] Xin, R. S. et al. “GraphX: A Resilient Distributed Graph System on Spark”. In: *First International Workshop on Graph Data Management Experiences and Systems - GRADES ’13* (2013), pp. 1–6. ISSN: 0002-9513. DOI: `10.1145/2484425.2484427`. arXiv: `1402.2394`.

[10] Zaharia, M. et al. “Discretized Streams: Fault-Tolerant Streaming Computation at Scale”. In: *Sosp* 1 (2013), pp. 423–438. DOI: `10.1145/2517349.2522737`.

[11] Armbrust, M. et al. “Scaling spark in the real world: performance and usability”. In: *Proceedings of the VLDB Endowment* 8.12 (2015), pp. 1840–1843. ISSN: 21508097. DOI: `10.14778/2824032.2824080`.

[12] *Sort Benchmark Home Page*. URL: `http://sortbenchmark.org/` (visited on 2017).

[13] Xin, R. et al. *GraySort on Apache Spark by Databricks*. Tech. rep. 2014.

[14] Wang, Q. et al. *NADSort*. Tech. rep. 2016, pp. 1–6.

[15] Visser, W. et al. “Model Checking Programs”. In: (2003), pp. 203–232.

[16] Pezzè, M. and Young, M. *Software testing and analysis: process, principles, and techniques*. Wiley, 2008. ISBN: 9780471455936.

[17] *NASA’s Ames Research Center*. National Aeronautics and Space Administration. URL: `https://www.nasa.gov/centers/ames/home/index.html` (visited on 2017).

[18] *Java PathFinder*. National Aeronautics and Space Administration. URL: `http://babelfish.arc.nasa.gov/trac/jpf/wiki` (visited on 2017).

[19] Havelund, K. and Pressburger, T. “Model checking JAVA programs using JAVA PathFinder”. In: *International Journal on Software Tools for Technology Transfer (STTT)* 2.4 (2000), pp. 366–381. ISSN: 14332779. DOI: `10.1007/s100090050043`.

[20] Gosling, James; Joy, Bill; Steele, Guy; Bracha, Gilad; Buckley, A. “The Java® Language Specification - jls8.pdf”. In: *Addison-Wesley* (2014), p. 688.

[21] Lindholm, T. et al. “The Java® Virtual Machine Specification”. In: *Managing* (2014), pp. 1–626.

# 5 Declaration of Academic Integrity

**Thesis Statement pursuant to § 22 paragraph 7 of APB TU Darmstadt**
I herewith formally declare that I have written the submitted thesis independently. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I clearly marked and separately listed all of the literature and all of the other sources which I employed when producing this academic work, either literally or in content. This thesis has not been handed in or published before in the same or similar form.
In the submitted thesis the written copies and the electronic version are identical in content.

Date: Signature:

## List of Figures

# List of Tables

# List of Listings

## Glossary

| Lineage | Lineage description |
|---|---|

## List of Abbreviations and Acronyms

| | |
|---|---|
| API | Application Programming Interface |
| HDFS | Hadoop Distributed File System |
| JPF | Java PathFinder |
| JVM | Java Virtual Machine |
| NASA | National Aeronautics and Space Administration |
| RDD | Resilient Distributed Dataset |
| SPF | Symbolic PathFinder |