Symbolic Execution of Apache Spark Programs

Omar A. Erminy Ugueto
Fachbereich Informatik
June 21, 2017

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: June 21, 2017

Supervisor: Prof. Dr. Guido Salvaneschi

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

1 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:

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

2 Introduction

2.1 Problem Statement

2.2 Illustrative Example

2.3 Hypothesis

2.4 Research Questions

3 Related Work

This chapter provides an overview to the main concepts and technologies related to our study; it aims to provide sufficient background information to fully understand the upcoming chapters. The reader is encouraged to skip this chapter if she is familiar with the concepts explained next. Section 3.1 introduces *Apache Spark* as the big data framework under study. Next, section 3.2 presents two program analysis techniques: explicit state model checking, and symbolic execution. Finally, section 3.3 gives an introduction to Java PathFinder (JPF) and its symbolic execution module.

3.1 Apache Spark

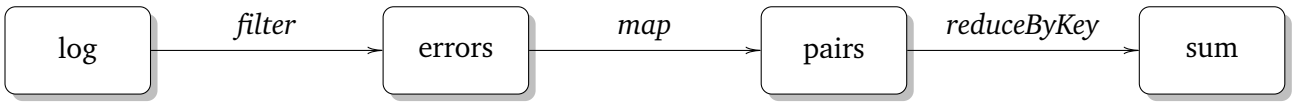
Spark is a distributed data processing framework that was first introduced in 2012 [43]. Similar to other systems, such as MapReduce [12] and Dryad [18], it aims to provide a clean and flexible abstraction to distributed computations on large datasets. However, Spark offers two advantages in comparison 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.

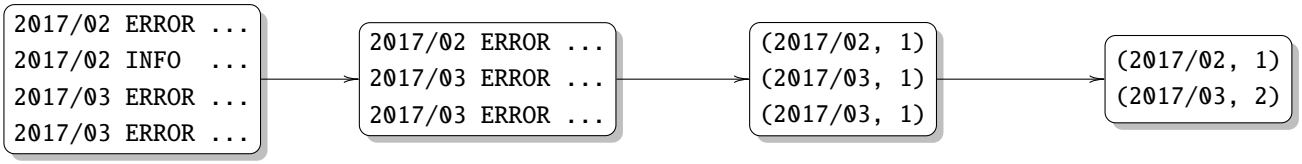
To interact with the RDD abstraction, Spark provides several APIs for different programming languages such as Java, Scala, Python and recently R [35]. Listing 3.1 presents a simple Spark program written with the Scala API, that processes log files in the search for errors. The operation in line 1 creates the first RDD from a log file, whose origin could be a local file or a partitioned file in a distributed file system such as

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



(a) Lineage of the program shown in listing 3.1. After each transformation, a new node in the lineage tree is created.



(b) Sample execution of the program shown in listing 3.1. If a task failed, Spark is capable to recalculate only the missing portions by retracing the operations in the lineage that led to the missing data.

Figure 3.1: Lineage and execution of the Spark program shown in listing 3.1. The lineage is independent from the association of an RDD to a variable; for example, the RDD resulting from the filter transformation is not assigned to a variable, however it is a node in the lineage tree.

Hadoop Distributed File System (HDFS) [39]. 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., a time stamp; 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.

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 3.1a depicts the resulting lineage of the program explained in listing 3.1 and figure 3.1b shows a sample execution of the same program.

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 [40, 3]. Several purpose-specific libraries are built on top of Spark, as is the case of: MLlib for machine learning [24], GraphX for graph computations [41], Spark Streaming for stream processing [44], and Spark SQL, an SQL-like interface for structured querying in Spark [4].

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 [32, 42, 38]. Overall, Spark offers a better performance in comparison to other data processing frameworks.

3.2 Program Analysis

Ensuring the quality and correctness of programs is a key aspect of the software development process. A wide variety of techniques are used to achieve this purpose, among which software testing is one of the most common. However, testing techniques are not always suitable; in particular, they are not effective detecting the causes of spurious failures that occur only under conditions that are hard to control or replicate (e.g., race conditions). Program analysis techniques result in a better approach in those scenarios because they reason about a model representing the system under test, thus, scoping down the program only to the relevant pieces that are used to verify a desired property.

In general, a model is an abstraction that preserves some selected attributes of an object or concept. They are used in many disciplines as mechanisms to improve communication and support decision making. Models that represent the execution of a program have to discard the unnecessary aspects of it while still preserving the capacity of explaining the potentially infinite execution states in a finite, compact, meaningful, and general view [26].

Programs can be modeled as a series of *states* that can be reached after certain actions occur, for instance, an action can be thought as the execution of code statements. In consequence, the *behavior* of a program can be defined as a sequence of states (or *path*) ranging from the beginning of the execution to its termination.

Control Flow Graphs (CFGs) are one example of such models, where nodes represent program statements and directed edges define the control flow relationship between them [1]. Listing 3.2 and figure 3.2 show a simple linear search algorithm and its corresponding Control Flow Graph respectively. CFGs serve as a starting point for different types of analyses, for example, data flow analysis where each node is augmented with information related to data accesses in order to verify that a variable is always initialized before is read.

Models like CFGs are useful when reasoning about properties related to the structure of the system under

```java
public static int search(int[] a, int elem) {
  for(int i = 0; i < a.length; i++) {
    if(a[i] == elem) {
      return i;
    }
  }
  return -1;
}
```

Listing 3.2: Linear search algorithm written in Java to illustrate the creation of a Control Flow Graph. If the element is contained in the array, the corresponding index is returned, otherwise a -1 is returned.

test. However, analyses of this kind often over-approximate on their conclusions given that they lack the means for conclusively asserting properties that depend on the execution of the program. In contrast, *explicit state model checking* and *symbolic execution* techniques reason about the properties of a program when this is being executed. The following sections discuss these two concepts in more detail.

3.2.1 Explicit State Model Checking

This technique, also known as Finite State Verification, consists of systematically exploring the potentially huge state space of a program in order to understand all possible executions. States are determined, for example, by all the possible values a variable or an expression can take during the execution of the system under test or by all possible interleavings that can result from the execution of a concurrent program.

As can be expected, the number of states for non-trivial programs grow exponentially; what is known as *state space explosion* problem. This condition poses several limitations to the practical use of the technique given that computational resources are quickly exhausted and timeliness conclusions are not feasible. Hence, the challenge relies on the reduction of the state space of the execution while still maintaining a full semantic correspondence between the model and the program, at least in terms of the property that is validated.

Strategies to make the state space smaller are frequently used when generating and exploring a model as an effort to make the technique applicable. For example, *Partial Order Reduction* is a strategy that aims to reduce the number of states to be explored by detecting when transitions resulting from concurrent operations result in equivalent states, making it necessary to explore such path only once.

Nonetheless, explicit state model checking proves itself useful because of its capacity to easily detect faults that would have been challenging, if not impossible, to notice with traditional software testing techniques. In particular, they result useful for discovering faults that would occur rarely under very specific conditions that cannot be generalized. They are commonly used to validate critical and concurrent systems, and are often combined with other testing techniques.

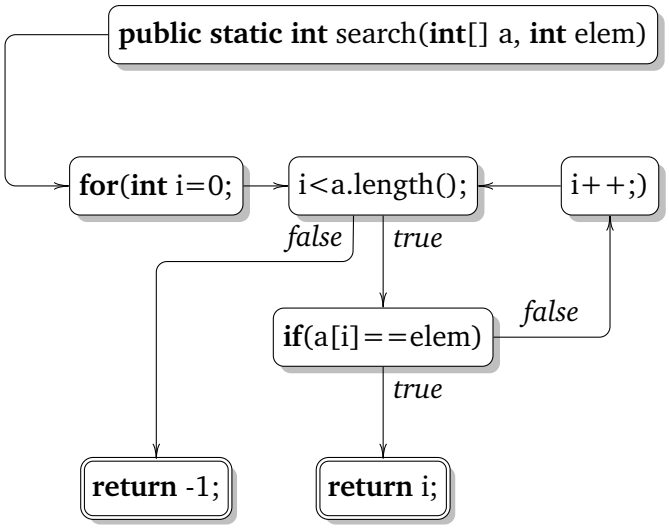


Figure 3.2: Control Flow Graph corresponding to the linear search algorithm shown in listing 3.2. The entry node is the signature of the method, while end nodes, represented with a double frame, contain the return statements that put at end to the execution. The *for* loop instruction was split into its composing statements to better display how the control flow work for this instruction.

3.2.2 Symbolic Execution

The first discussions of symbolic execution date several decades back [17, 21]. The idea consists of executing a program using a set of "symbolic" input parameters in order to build logical predicates that characterize all possible executions. This symbolic parameters can be thought as mathematical variables, in contrast to what would be a concrete value. Throughout the execution, the symbolic values are operated, which generates more complex symbolic expressions. Moreover, control flow statements define logical predicates that could depend on symbolic expressions, bridging the representation of the program from its operational view to a series of logical expressions.

Conditional statements are trivial to evaluate when tracing the execution of a program with a concrete value; the branching conditions are simply evaluated and a path is chosen to proceed with the execution. However, if the branch condition depends on symbolic values, both paths corresponding to the *true* and *false* evaluation respectively are an option, hence, the execution continues to be traced through both branches. As a result, each execution path of the program is characterized by a sequence of predicates and how they were evaluated; also known as path condition.

A path condition is satisfiable if there exists a group of concrete input values that makes its logical predicate hold, which means that these values can steer the execution of the program through that path. Whereas, if the path condition cannot be satisfied then it will be impossible for any concrete execution to follow that path, rendering the path infeasible. Interestingly enough, each satisfiable path condition represents an equivalence class of concrete input values. Figure 3.3 shows the symbolic execution tree of the program in listing 3.3.

Symbolic execution could be combined with pre-conditions, post-conditions, loop invariants and, in general, any assertion at any given point in the source code. Comparing path conditions against these vali-

```
1 public void trivial(boolean a, int b, boolean c) {
2   int x = 0, y = 0, z = b + 1;
3   if (a) { x = -1; }
4   if (z > 5) {
5     if (!a && c) { y = -1; }
6   }
7   assert x + y != 0;
8 }
```

Listing 3.3: Trivial program to illustrate how symbolic execution works.

dations help reasoning about the status of the execution, in particular when detecting faulty programs. Moreover, loop invariants are helpful when executing loops symbolically, given that in most cases loops lead to unbounded chains of logical predicates due to the inability to evaluate the stopping condition concretely.

To determine if a path condition is satisfiable, symbolic execution tools make use of constraint solvers and theorem provers. Though having improved considerably in recent years, solvers and provers still represent the main bottlenecks for the application of symbolic execution in large scale programs [8].

Although full verification based on symbolic execution might be unfeasible, reduced domains and specific validations could benefit from its principles. For example, there are several applications for symbolic execution in program analysis; the most common are input data generation [9], test case generation [7, 10, 14, 36] and static detection of errors [6, 34], among many others [11, 31].

3.3 Java PathFinder

Developed at NASA's Ames Research Center [25], Java PathFinder (JPF) is an execution environment for verification and analysis of Java bytecode programs [37, 19]. Since its publication in the year 2000 [16], 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 [15]. 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 JVM specification. A JVM implementation serves as an interpreter of Java bytecode and allows the optimization and execution of the program tailored for the host platform [22].

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

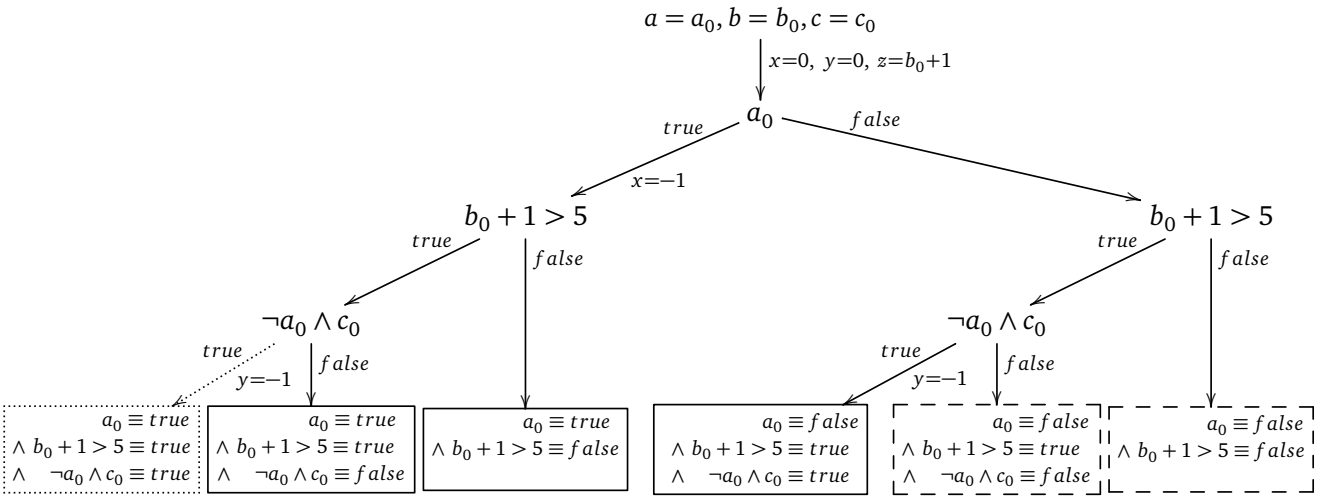


Figure 3.3: Symbolic execution tree of the program presented in listing 3.3. The root node represents the input parameters as symbolic values while the intermediate nodes illustrate the control flow of the program evaluated on these symbolic values wherever possible.Each intermediate node branches into two options, each corresponding to the predicate evaluating to *true* or *false* respectively. Also, each branch is labeled with the statements executed following one of the evaluations. More importantly, the leaves collect the predicates that form the path condition for that particular execution. The left most leave in the dotted frame contains an unfeasible path condition given that the predicate cannot be satisfied. Moreover, the path conditions in the dashed frames define executions that will fail the assertion of line 7.

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 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 JVM; in other words, a JVM on top of a 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 3.4 portrays the components 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,

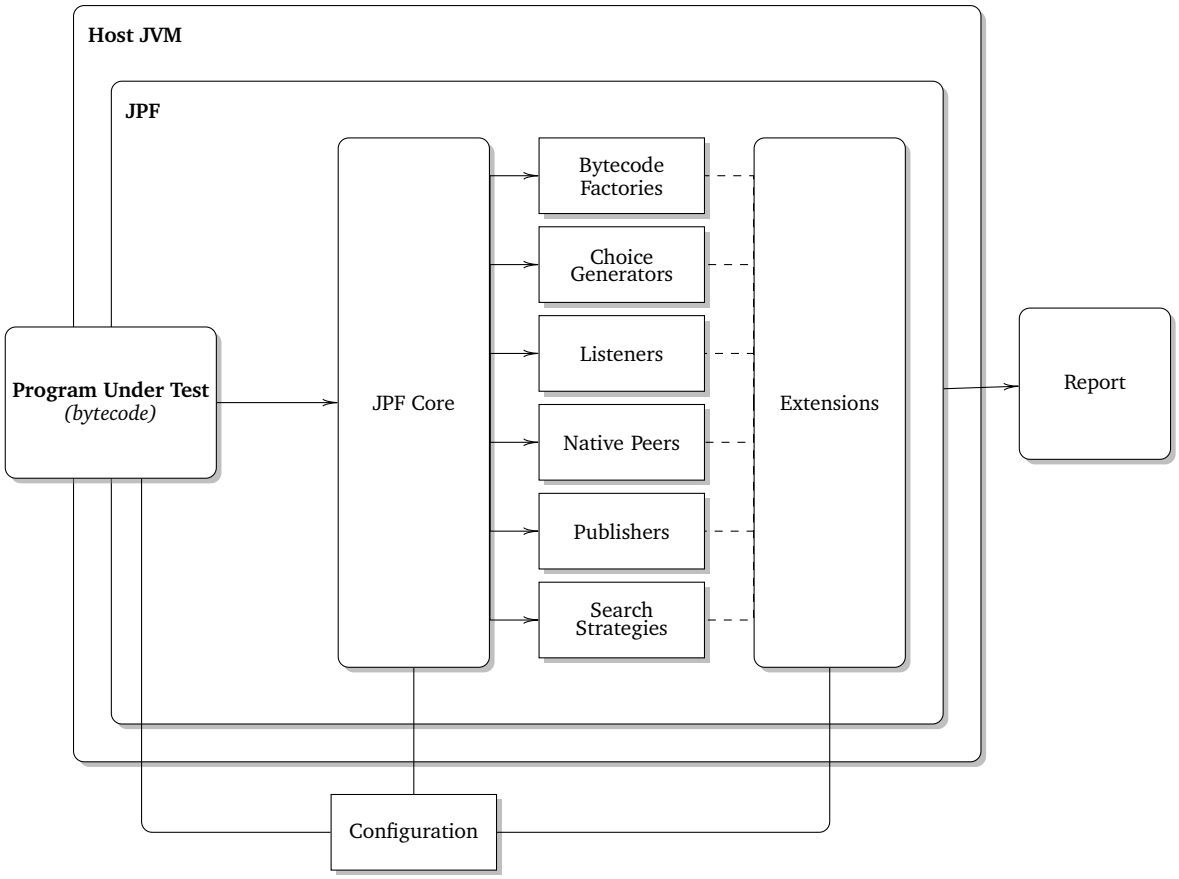


Figure 3.4: JPF Components and workflow. The program under test, taken as bytecode, is loaded into the JPF Virtual Machine which is un turn executed on top of the host JVM. Libraries used in the program under test need to be visible to the core in order to be able to execute the program correctly. Note that the core is comprised by several components in charge of directing the execution of the analysis. The behavior of this components could be extended by including modules. The final output is a report in its general sense; this could range from simple console output to automatic test generation. Moreover, several configuration inputs dictate how the participants proceed through their execution.

JPF backtracks to a recorded state in order to explore a new path.

Listing 3.4 introduces an example that illustrates better how 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 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 could start exploring all different possible combinations spanning the domain of all integer values that can be represented, 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, defining a minimal range of integers that could actually occur in an execution; in this case ranging from 0 to the parameter passed to the *nextInt* function. Consequently, a combination that triggers the invalid

```
1  import java.util.Random;
2
3  public class RandomExample {
4    public static void main(String[] args) {
5      Random random = new Random();
6      int a = random.nextInt(2);
7      int b = random.nextInt(3);
8      int c = a/(b+a-2);
9    }
10 }
```

Listing 3.4: 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 [25])

operation is found promptly and reported back to the user. Figure 3.5 depicts how JPF explores the state space in order to validate the program.

A key aspect of JPF was to make it extensible and customizable. Following a modular design, users of the tool are capable of tuning JPF up to the needs of a wide variety of analyses and verifications. Its main components 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

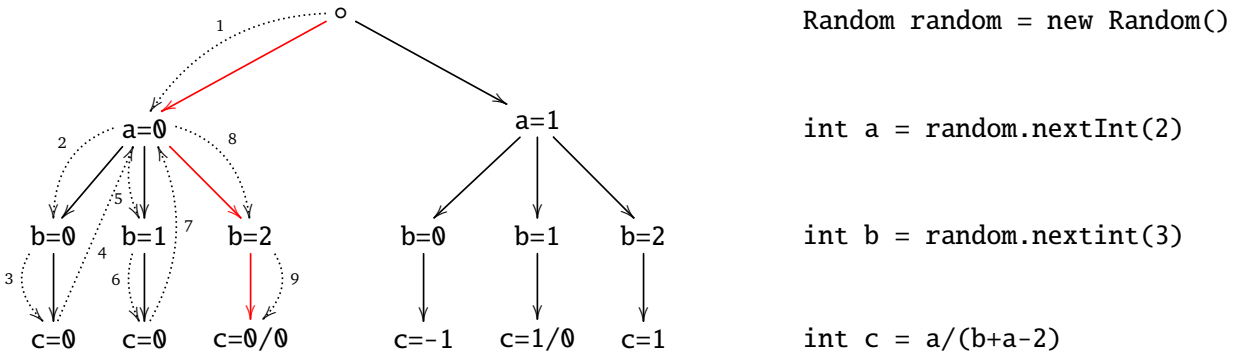


Figure 3.5: State space exploration of the program shown in listing 3.4. JPF starts checking the state space whenever the the conditions that trigger the property to be validated are found; in this case, using random values. The `nextInt` instruction causes JPF to register a *Choice Generator* and start exploring the state space of the possible options. The dashed edges represent the search strategy used to explore the state space; in this case depth-first search. If a given execution path gets to an end and no unexpected behavior is found, JPF backtracks to the latest instruction where a *Choice Generator* was registered and tries a different value. The red arrows point to an execution that triggers an error. Whenever an error is found JPF halts the validation and reports its findings.

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 be 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.

3.3.1 Symbolic PathFinder

As one of the earliest extension modules, Symbolic PathFinder (SPF) integrates symbolic execution principles into JPF . Although it has undergone several modifications throughout the years [20, 29, 2], its current mode of operation consists of replacing the concrete execution semantics of the default JPF model checker with a corresponding symbolic interpretation [28]. In recent years SPF has had some improvements, primarily supporting Java 1.7 and better detection of unfeasible paths [23].

The introduction of symbolic semantics is achieved through the use of the *SymbolicInstructionFactory* class;

an extension to the default bytecode instruction set that interacts with symbolic values and expressions. For example, operating two symbolic integers using the *IADD* bytecode instruction results in the creation of a symbolic expression that represents the sum of those integers. Furthermore, symbolic values and expressions are assigned to variables and fields, instead of the corresponding concrete representation that would result from a normal execution.

SPF supports symbolic operations on several primitive types: booleans, integers and doubles, as well as on complex data structures. Nevertheless, only limited support to symbolic *String* operations is offered in the latest SPF version.

The interpretation of branching instructions is a key point of symbolic execution because it determines how the subsequent paths ought to be explored. SPF process branches by generating a special choice generator called *PCChoiceGenerator* every time a conditional instruction is found. The choices registered by the choice generator correspond to the evaluation of the predicate and its negation respectively, where each choice is linked with a path condition reflecting how the predicate was evaluated. SPF takes advantage of JPF 's model checking framework to explore the symbolic state space by considering only the registered choices at branching instructions.

SPF checks the satisfiability of path conditions using third-party constraint solvers like Choco [27], CORAL [33], and CVC3 [5]. Most of these solvers are geared towards solving complex numerical constraints, while solving structural constraints (like *String* predicates) are limited at best or incompatible at worst. If a path condition is unsatisfiable, SPF backtracks to the latest branching point and tries out a different choice.

Listeners are used to gather information during the evaluation of path conditions. Publishers make use of this information to present it to the user in different ways: One common case is to partition the input data in the different equivalence classes, while other is the automatic generation of unit tests. Using symbolic execution to automatically generate a test suite with path coverage is a research topic explored in several studies [30, 36, 14].

Configuration files are used to indicate which methods should be executed symbolically and also specify which their parameters are to be considered as symbolic or concrete values. By combining symbolic and concrete values during the execution of a method marked to be symbolically executed, SPF provides the framework for concolic execution [13].

4 JPF-SymSpark

JPF-SymSpark is a JPF module whose goal is to coordinate the symbolic execution of Spark programs and produce a reduced input dataset that ensures full path coverage on a regular execution. It builds on top of SPF to delegate the handling symbolic expressions while it focus on how to interconnect Spark's transformations and actions in order to reason coherently over the program's data flow.

In the following sections we will provide an overview of the logical process carried out by the analysis and an explanation of the different components that conform the module.

4.1 Logic

A Spark program consists of a chain of transformations on one or more RDDs to finally conclude with an action. RDDs are manipulated through an API that provides the general guidelines on how the data collection is to be processed without falling into the specifics. For example, the *filter* transformation indicate that only the elements matching a given filtering condition would be selected, without specifying exactly what is the condition to be evaluated. A similar approach is followed by most of the actions and transformations in Spark.

The precise behavior of most actions and transformations is defined by the programmer. Given that most of Spark's actions and transformations are first order functions (they accept a function as a parameter), the programmers define a custom function that fulfills the contract of the specific operation. For example, again the *filter* transformation expects as a parameter a function that takes an element of the same type as the type of the elements in the collection handled by the RDD and returns a boolean value. When the *filter* transformation is later invoked, it calls the passed function with each element in the RDD and, depending on the output, it decides if the value is filtered or not.

Having this in mind, the symbolic execution of an isolated Spark operation depends solely on the behavior of the function passed by the user. However, when analyzing a whole program, special considerations for every particular operation must be taken into account. These considerations are different in nature but mostly refer to how output values are percolated to the subsequent operations in order to ensure the correct analysis of the next functions.

The whole process could be summarized in the following three steps:

1. Identify the Spark operation

2. Carry out the symbolic execution of the passed function

3. Take special considerations based on the executed Spark operation

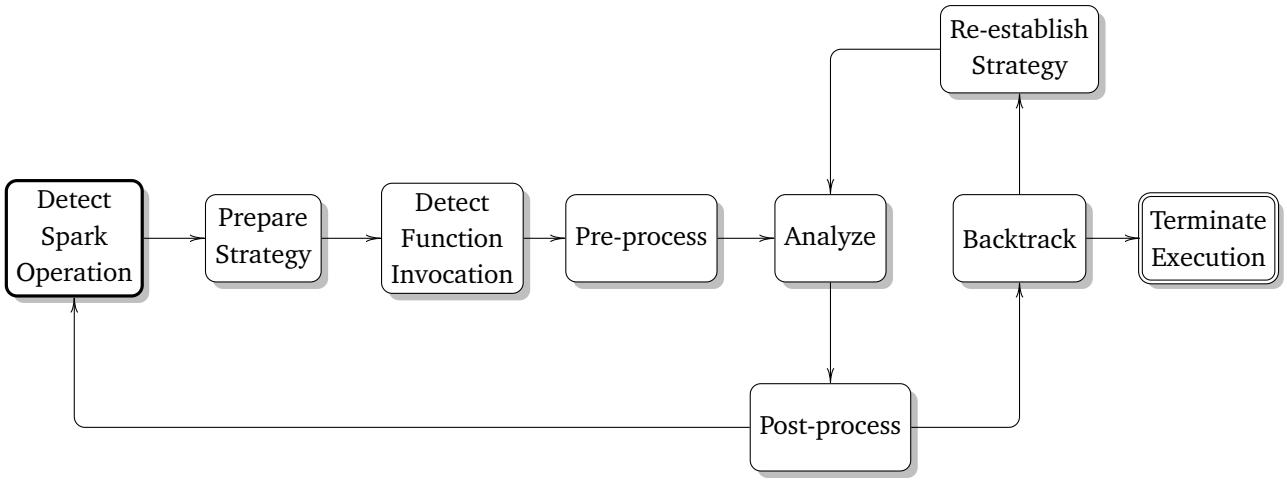


Figure 4.1: State diagram of the symbolic execution process of Spark programs.

Following these three steps, the symbolic execution of a Spark program using SPF as the underlying analysis framework is represented by the state diagram depicted in 4.1. Here we consider how a black-box analysis should proceed in order to reason about the execution flow of a Spark program and the control flow instruction that might occur in them.

The starting point of the analysis consists in detecting that a Spark operation of interest is being executed; relevant operations can be defined by the user beforehand. Once this has happened, the next step is to prepare for the exact operation that was detected, for example, indicating what function was passed to the detected operation and prepare the SPF analysis to consider its inputs as symbolic. Generally, these two steps occur simultaneously but given their semantic differences in the process, it was important to highlight them as different states of the analysis.

The real analysis begins once the passed function is invoked. The process is split in three stages: *pre-process*, *analysis* and *post-process*. During the *pre-process* stage, we must ensure that the parameters passed to the function are correctly instantiated; for example, if a *map* and a *filter* transformations took place in that order, we must ensure that the input symbolic expression passed to the function executed by the *filter* is the output of the function invoked by the *map* transformation. This guarantees a coherent inter-methodical analysis. The subsequent stage is the core analysis, which proceeds in the same way as an analysis of a regular method in SPF would do. Lastly, during the post-process stage the framework makes all the necessary preparations to be able to continue the symbolic execution of subsequent Spark operations.

Once the analysis of a Spark operation is done, the framework continues to explore the program. This can lead to the detection of another relevant Spark operation. Finally, once the execution has finished, JPF will backtrack to any decision points defined by the *Choice Generators*. These points always take place inside one of the functions passed to any Spark operations; this is why the framework has to re-establish a strategy corresponding the Spark operation containing the invoked function. The analysis continues as usual once the strategy has been re-established. After all choices have been explored the execution terminates and the analysis provides an output.

4.1.1 A Concrete Example

The trivial example presented in listing 4.1 depicts a simple Spark program with no purpose in itself. However, this simple example allows us to explain better how the analysis will be carried out. The relevant Spark operations in this example are the *map* and *filter* transformations in lines 10 and 14 respectively. All other operations related to Spark are not relevant.

```
1  SparkConf conf = new SparkConf()
2  .setAppName("Example")
3  .setMaster("local");
4
5  JavaSparkContext spark = new JavaSparkContext(conf);
6
7  List<Integer> numberList = Arrays.asList(1,2,3);
8  JavaRDD<Integer> numbers = spark.parallelize(numberList);
9
10 numbers.map(v1 -> {
11   if(v1 > 1) return v1;
12   else return v1+2;
13 })
14 .filter(v2 -> v2 > 2);
15
16 spark.stop();
17 spark.close();
```

Listing 4.1: Trivial example to illustrate the symbolic execution of spark programs. The program itself has no real purpose other than to serve as a good scenario to demonstrate inter and intra procedural conditions of the analysis.

The first operation detected during the analysis is the *map* transformation in line 10. At this point, *JPF-SymSpark* approaches the situation following a "map" strategy and prepares itself for the imminent invocation of the function passed to the *map*. All the functions in this example are depicted as lambda functions. Once the function is invoked, the framework proceeds with the pre-processing stage, however, because no previous operations were executed, the initial input for the function is a trivial symbolic reference (V_0). This reference serves as our symbolic variable for determining the possible input values that would explore all the feasible paths in the program.

During the symbolic execution of the function we find that there is a branching instruction in line 11. This represents a decision point and, for this reason, a *choice generator* is registered with two options: one where V_0 is greater than one and another where it is less than or equals to one. The control flow continues with one of the paths and stores the other for a later exploration. Given the nature of the *map* transformation, the input parameter might suffer a certain transformation which, in turn, is the returned value, as it is shown in line 12. During the pre-processing of the operation the symbolic expression $V_0 + 2$ is then set to be the input value of the immediate Spark operation, which in this case is a *filter*. The *filter* transformation is processed in a similar fashion except that in this case the input value used in its function is instantiated to whatever output generated the *map* transformation.

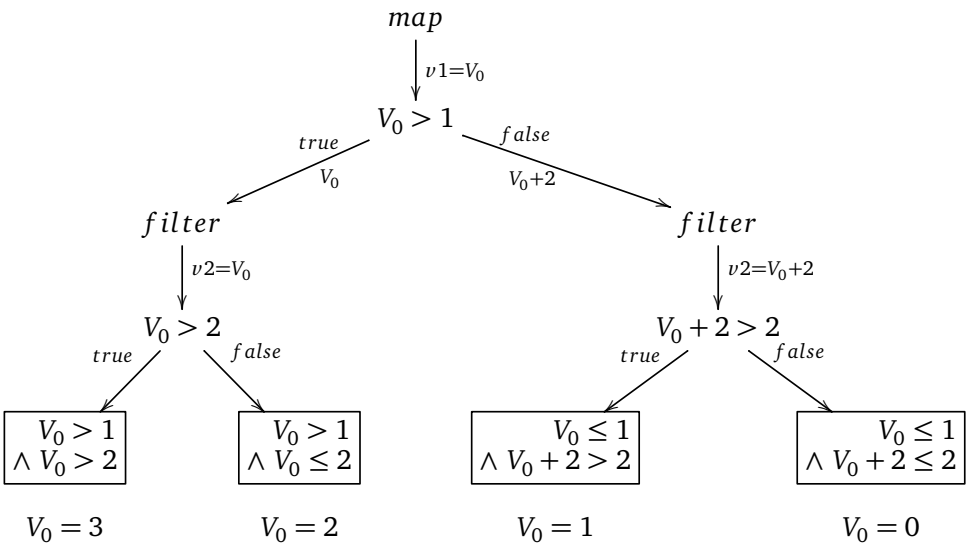


Figure 4.2: Symbolic execution tree corresponding to the Spark program shown in listing 4.1. The input set $\{3, 2, 1, 0\}$ represents a minimal input set that would explore all feasible paths in the program.

The function passed to the *filter* transformation returns a boolean that depends on the symbolic input value. Given the nature of this kind of instruction, SPF registers a *choice genertor* in order to explore the possible outcomes of evaluating the boolean condition. Again, one of the paths is chosen and the analysis continues. At this point there are no more relevant Spark operations and the execution comes to an end, thus, triggering a backtrack to the last unexplored path. Finally, the analysis continues until there are no more unexplored paths left.

To further illustrate the example, figure 4.2 shows the symbolic execution tree of the program. One interesting aspect to note is that the results of the *map* are percolated to the subsequent Spark operations; such is the case of the rightmost subtree in the symbolic execution tree.

When observed in this way, the analysis of the program turns out to be similar to the sequential execution of the respective input functions of each of the Spark operations in the program. However, this is not always the case given that some operations have particular semantic implications, for example, *flatMap* produces multiple symbolic output, hence, making it impossible to simply connect the function passed to a *flatMap* as it is to any following operation.

After the analysis is done, the module can solve the resulting path conditions and obtain a representative value in the range to produce a reduced input data set that is able to offer full path coverage of the program.

4.2 Module

This section describes general structure and technical aspects of the *JPF-SymSpark* module. The work presented here is based on the logical processes defined in the previous section. The following sections explain the different components that conform the module and what role do they play in the whole analysis.

4.2.1 Spark Library and JPF

When executing an analysis using JPF , the whole program is run under an instrumented JVM that keeps track of the execution state of the program. JPF considers every program statement executed, even if it is executed by third-party libraries or dependencies indirectly invoked in the system under test. These libraries must be included in the JPF 's classpath (which is a different classpath than the normal Java classpath of the system under test) if they are executed, because if not, JPF will fail indicating that it is not able to find certain references during execution.

On the other hand, Spark, as many other modern applications, depends on a constantly growing number of external libraries. To execute an analysis on a Spark program with JPF one could include all this libraries and dependencies in the the JPF 's classpath and let tool handle all the invocations internally. However, this approach has several problems: First, the execution of more statements increases the workload and state space of JPF . Second, some of Spark's operations handle native calls that, for example, deal with the way tasks are placed in the OS; JPF does not handle such native operations by default, which leads to the need of creating surrogate peers that mock the behavior of such calls. Lastly and more importantly, most of these operations are called in methods that are unrelated to the actions and transformations that are relevant to the symbolic execution, leading to an unnecessary overhead that does not provide any benefit.

Because of all these reasons, including the Spark library and all its dependencies was not a reasonable approach. Instead, we decided mock up the Spark library, in order to mimic some of the classes that participate in a Spark program. The idea is to minimize the number of external dependencies and native calls while at the same time replacing the implementations of methods irrelevant to the analyses with simplified versions of themselves. Listing 4.2 is an example of how a class that is irrelevant to the analysis is simplified. In the regular Spark library the *JavaSparkContext* class triggers a lot of heavy processes, like initializing the whole Spark framework; now it is just reduced to empty or simple code blocks.

However, some of the methods invoked by the Spark library are relevant to the analysis. Such is the case of the methods defined in the *JavaRDD* class and the rest of the classes in the RDD family. These methods include operations like *filter*, *map* and *reduce*, that make use of the functions passed by the programmers. In these cases, it is extremely relevant that the passed function is invoked inside these methods so the analysis can be triggered following the usual SPF approach. Listing 4.3 shows an example of the mocked *filter* method of the *JavaRDD* class. The function passed to the *filter* method is invoked with the first element of the RDD only and the returned value is the current RDD itself given that it does not affect the end result when using symbolic input parameters.

```
1  import java.util.Arrays;
2  import java.util.List;
3  import org.apache.spark.SparkConf;
4
5  public class JavaSparkContext {
6    public JavaSparkContext(SparkConf conf){}
7    public void stop() {}
8    public void close() {}
9    public <T> JavaRDD<T> parallelize(List<T> list) {
10     return new JavaRDD<T>(list);
11   }
12   public JavaRDD<String> textFile(String file) {
13     return new JavaRDD<String>(Arrays.asList(""));
14   }
15 }
```

Listing 4.2: Mocked version of the *JavaSparkContext* class. The methods are as simple as they could be while still maintaining the contract of the original class. Note that the classes *SparkConf* and *JavaRDD* are also mocked.

```
1  public JavaRDD<T> filter(Function<T,Boolean> f) {
2    try {
3      f.call(list_t.get(0));
4    } catch (Exception e) {
5      e.printStackTrace();
6    }
7    return this;
8  }
```

Listing 4.3: Mocked filter method in the JavaRDD class. The function passed to the method is invoked. Note that the *Function* interface is also mocked.

The mocked up Spark library is already included into the dependencies of the *JPF-SymSpark* module. Nevertheless, the implementation is not extensive, which might require further expansion as the different cases and programs require. Moreover, the mocked up library is bound to version 2.0.2 of the original Spark library, which poses a drawback in terms of consistency if the core behavior of Spark changes in future versions.

Having effectively discarded irrelevant portions of the system under test by the means of the mocked up library, it is simpler to identify the relevant Spark operations that have an impact on the analysis.

4.2.2 Instruction Factory

The first step for carrying out the analysis is to determine when a Spark operation is being executed. Given that all relevant operations are implemented as non-static methods in the concrete *JavaRDD* class, the bytecode instruction of interest is *invokevirtual*. This instruction is in charge of dispatching Java methods, unless they are interface methods, static methods or some other special cases (*invokeinterface*, *invokestatic* and *invokespecial* are used respectively in these cases) [22].

For this purpose, we implemented the *SparkSymbolicInstructionFactory* class; which extends from the *SymbolicInstructionFactory* class defined in the SPF module. The goal of this class is to solely intercept calls to the *invokevirtual* bytecode instruction and validate if they intend to dispatch one of the Spark operations relevant to the analysis.

Just to illustrate this situation better in the case of the Java implementation, let us assume that the *filter* transformation is being called on an existing RDD such as

```
rdd.filter(...)
```

then, the corresponding bytecode will look like the following

```
invokevirtual PATH/JavaRDD.filter:(PATH/Function;)PATH/JavaRDD;
```

with "PATH" representing the full package path where the classes or interfaces are located. The rest of the instruction represents the method name and the method descriptor; sufficient information for identifying the methods of interest. The function parameter was intentionally omitted because, although the passed parameter must implement the *Function* interface, this can be done in several ways; being the most common lambda functions and anonymous classes. At this point, neither of this two approaches represent a difference when detecting the Spark operation, however, it will require special attention later on when detecting the invocation of the passed function.

4.2.3 Spark Listener and Flow Coordinator

4.2.4 Method Strategies

4.2.5 Choice Generators

5 Evaluation

6 Future Work

6.1 Limitations

The limited support to String symbolic operations. The lack of a solver that specializes on Strings makes it limited to supporting many big data tasks.

The complications when dealing with lambda expressions makes it difficult to use the tool on real spark applications. Because most current developers prefer java8 syntax favoring its flexibility and reduced verbosity.

The ugly ill-maintained codebase of jpf-symbc makes it cumbersome to extend and easily outdated due to abundance of code smells and bad practices.

Although support for other solvers is mentioned, in practice, many of them fail due to missing libraries which are outdated when independently included.

Limited support to objects and how symbolic objects are created and shared

Limited support to String operations

Bibliography

[1] Allen, F. E. “Control Flow Analysis”. In: *Proceedings of ACM Symposium on Compiler Optimization* (1970), pp. 1–19. ISSN: 03621340. DOI: 10.1145/800028.808479.

[2] Anand, S., Păsăreanu, C. S., and Visser, W. “JPF–SE: A Symbolic Execution Extension to Java PathFinder”. In: *Tacas 2007* (2007), pp. 134–138. ISSN: 03029743. DOI: 10.1007/978-3-540-71209-1.

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

[4] 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.

[5] Barrett, C. and Tinelli, C. “CVC3”. In: *Proceedings of the 19th International Conference on Computer Aided Verification*. CAV’07. Berlin, Germany: Springer-Verlag, 2007, pp. 298–302. ISBN: 978-3-540-73367-6.

[6] Bush, W. R., Pincus, J. P., and Sielaff, D. J. “A static analyzer for finding dynamic programming errors”. In: *Software Practice and Experience* 30.November 1998 (2000), pp. 775–802.

[7] Cadar, C., Dunbar, D., and Engler, D. R. “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs”. In: *Proceedings of the 8th USENIX conference on Operating systems design and implementation* (2008), pp. 209–224. ISSN: <null>. DOI: 10.1.1.142.9494.

[8] Cadar, C. and Sen, K. “Symbolic execution for software testing: three decades later”. In: *Communications of the ACM* 56.2 (2013), pp. 82–90. ISSN: 0001-0782. DOI: 10.1145/2408776.2408795.

[9] Clarke, L. a. “A System to Generate Test Data and Symbolically Execute Programs”. In: *IEEE Transactions on Software Engineering* SE-2.3 (1976), pp. 215–222. ISSN: 0098-5589. DOI: 10.1109/TSE.1976.233817.

[10] Csallner, C. and Fegaras, L. “New Ideas Track : Testing MapReduce-Style Programs Categories and Subject Descriptors”. In: ().

[11] Csallner, C., Tillmann, N., and Smaragdakis, Y. “DySy: dynamic symbolic execution for invariant inference”. In: *Proceedings of the 13th international conference on Software engineering - ICSE ’08* (2008), p. 281. ISSN: 02705257. DOI: 10.1145/1368088.1368127.

[12] 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.

[13] Godefroid, P., Klarlund, N., and Sen, K. “DART: directed automated random testing”. In: *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation* (2005), pp. 213–223. ISSN: 03621340. DOI: 10.1145/1065010.1065036.

[14] Godefroid, P., Levin, M. Y., and Molnar, D. a. “Automated Whitebox Fuzz Testing”. In: *Ndss* July (2008). ISSN: 1064-3745.

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

[16] 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.

[17] Hoare, C. A. R. “An Axiomatic Basis for Computer Programming”. In: *Communications of the ACM* 12.10 (1969), pp. 576–580. ISSN: 00010782. DOI: 10.1145/363235.363259.

[18] 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.

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

[20] Khurshid, S., Păsăreanu, C. S., and Visser, W. “Generalized Symbolic Execution for Model Checking and Testing”. In: (2003), pp. 553–568.

[21] King, J. C. “Symbolic execution and program testing”. In: *Communications of the ACM* 19.7 (1976), pp. 385–394. ISSN: 00010782. DOI: 10.1145/360248.360252.

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

[23] Luckow, K. S. and Păsăreanu, C. S. “Symbolic PathFinder V7”. In: *SIGSOFT Softw. Eng. Notes* 39.1 (2014), pp. 1–5. ISSN: 0163-5948. DOI: 10.1145/2557833.2560571.

[24] 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.

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

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

[27] Prud'homme, C., Fages, J.-G., and Lorca, X. *Choco Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S. 2016.

[28] Păsăreanu, C. S. and Rungta, N. “Symbolic PathFinder: Symbolic Execution of Java Bytecode”. In: *25th IEEE/ACM International Conference on Automated Software Engineering* 2 (2010), pp. 179–180. DOI: 10.1145/1858996.1859035.

[29] Păsăreanu, C. S. and Visser, W. “Symbolic Execution and Model Checking for Testing”. In: *HVC 2007* 4424.2007 (2008), pp. 17–18.

[30] Păsăreanu, C. S. et al. “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 ISSTA 08* (2008), pp. 15–26. DOI: 10.1145/1390630.1390635.

[31] Siegel, S. F. et al. “Using model checking with symbolic execution to verify parallel numerical programs”. In: *Proceedings of the 2006 international symposium on Software testing and analysis* (2006), pp. 157–168. DOI: 10.1145/1146238.1146256.

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

[33] Souza, M., Borges, M., and Corina, S. “CORAL: Solving Complex Constraints for Symbolic PathFinder”. In: ().

[34] Tomb, A., Brat, G., and Visser, W. “Variably interprocedural program analysis for runtime error detection”. In: *Proceedings of the 2007 international symposium on Software testing and analysis ISSTA 07* January 2007 (2007), p. 97. DOI: 10.1145/1273463.1273478.

[35] 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.

[36] Visser, W., Păsăreanu, C. S., and Khurshid, S. “Test Input Generation with Java PathFinder”. In: *ACM SIGSOFT Software Engineering Notes* 29.4 (2004), p. 97. ISSN: 01635948. DOI: 10.1145/1013886.1007526.

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

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

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

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

[41] 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.

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

[43] 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.

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

List of Figures

List of Tables

List of Listings

A Appendix - Collaborations to SPF

A.1 Detection of Synthetic Bridge Methods

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.

A.2 Order of String Path Conditions

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.

B Appendix - Installation and Use

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.