Symbolic Execution of Higher-order Functions in Big Data Systems

Symbolische Ausführung von Funktionen höherer Ordnung in Big Data Systemen

Master-Thesis von Omar A. Erminy Ugueto

Tag der Einreichung: August 12, 2017

1. Gutachten: Prof. Dr. Guido Salvaneschi
2. Gutachten: M.Sc. Pascal Weisenburger

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik
Reactive Programming Technology

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

Master Thesis
Symbolic Execution of Higher-order Functions in Big Data Systems
Symbolische Ausführung von Funktionen höherer Ordnung in Big Data Systemen

Submitted: August 12, 2017

Gutachten 1: Prof. Dr. Guido Salvaneschi
Gutachten 2: M.Sc. Pascal Weisenburger

Prof. Dr. Guido Salvaneschi
Fachgebiet Softwaretechnik
Fachbereich Informatik
Technische Universität Darmstadt
Hochschulstraße 10
64289 Darmstadt

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

The last twenty years have brought many technological advancements. The massive worldwide adoption of the Internet has created a gigantic platform where uncountable amounts of information are produced and shared. Likewise, the shift of many industries into the digital plane enabled the generation and storage of huge volumes of data. Processing such volumes of data in a sensible and timely manner required innovative computational paradigms; this led to the creation of several successful distributed models that proved to be better suited for tasks of such magnitude. One example is Apache Spark, a fault-tolerant distributed processing framework that uses a shared memory abstraction in order to offer better performance.

However, as any other software, applications written for these big data frameworks are susceptible to bugs and errors. The difference in this case is that bugs in distributed software have other implications in comparison to the traditional single-node computing. For instance, mishandled errors can permeate to multiple nodes, potentially collapsing a whole cluster of computers. Furthermore, if the faulty distributed computations are executed in third-party cloud servers, monetary losses could be inflicted due to unnecessary use of the resources and service downtime.

Although several program testing techniques have been ported to the context of distributed programming and have been the subject of research studies, program analysis approaches have received less attention both in the industry and the academia. Formal methods and code analyses could also prove useful towards the goal of improving code quality and their automated nature could provide a mechanism for a continuous evaluation.

This work aims to clarify the applicability of model checking techniques, in particular symbolic execution, in the context of big data systems. For this purpose, it introduces *JPF-SymSpark*, a symbolic execution framework for Apache Spark programs built as an extension of Java PathFinder. The main goal of *JPF-SymSpark* is to generate reduced input datasets that offer full path coverage of the analyzed program. Such datasets can have several uses in the development process of a Spark application, for example, as input data for unit tests. The tool is able to symbolically execute Spark programs that handle primitive data types and Strings as their input datasets. It is also capable of chaining multiple Spark operations during a symbolic execution; providing the mechanisms for a complete analysis of a program.

The evaluation of the approach is twofold. First, a qualitative evaluation that aims to determine how compliant *JPF-SymSpark* is in terms of a series of functional and non-functional requirements defined for a tool of such purpose. Second, a performance assessment of the iterative symbolic operations carried out during an analysis. Additionally, a discussion over the limitations of the approach is presented in order to establish the scope and capabilities of its current state. Furthermore, the study concludes with a general discussion on the applicability of model checking techniques in the context of Java big data systems based on the currently available tools and frameworks.

Contents

1 Introduction

1.1 Background

The last twenty years have brought many technological advancements. The massive worldwide adoption of the Internet has created a gigantic platform where uncountable amounts of information are produced and shared. Likewise, the shift of many industries into the digital plane enabled the generation and storage of huge volumes of data. In 2012, Google stated that its search engine had indexed more than 30 trillion unique URLs and that it had handled more than one trillion searches that year [21]. Similarly, Rolls Royce announced they were using big data to drive the manufacturing process of their engines, gathering as much as three petabytes of data per year for every single piece conforming a turbine [36, 16].

Processing such volumes of data in a sensible and timely manner required innovative computational paradigms. Google's GFS [18] and MapReduce [15] were presented as a successful model for distributed computation that was flexible enough to handle a wide variety of big data related tasks while at the same time it was able to scale swiftly with the amount of data processed. One of the key elements that fostered the success of MapReduce among developers is that the complex logic of distributed software was hidden behind programming constructs acting as abstractions; the outcome was a flexible but bounded toolkit for the development of distributed applications.

Ever since the rise of MapReduce, the attention of both the industry and the academy has turned into distributed computation algorithms in the search of better and more efficient approaches. This gave rise to many improved models and big data development frameworks; one of them is Apache Spark [63], a fault-tolerant distributed processing framework that uses a shared memory abstraction in order to offer better performance.

1.2 Problem Statement

As any other software, applications written for big data frameworks are susceptible to bugs. However, a bug in an application that runs in a distributed mode has different implications in comparison to the traditional single-node computing. For instance, mishandled errors can permeate to multiple nodes, collapsing a whole cluster of computers. Furthermore, if the distributed computations are executed in third-party cloud servers, faulty applications could implicate additional monetary costs to the party paying for the cloud platform. This implications also affect the development process, a developer should not check the applications only in a cloud environment; she should count with the right tools to assess and ensure the quality of the software before deploying it into the cloud.

For these reasons, ensuring code quality during the development phase of big data applications is critical. In this sense, several program testing techniques have been used in the context of distributed programming. Unit testing has been widely adopted without major adaptations, given that many of these frameworks are based on existing programming languages that supported the testing technique. Moreover, other techniques like debugging have been adapted to match the specifics of big data applications. Tools like Titian [26] and BigDebug [23] focused on debugging capabilities that enforced interactivity and data provenance, being the latter a particular relevant matter in data intensive applications.

However, program analysis techniques have received less attention. At most, some big data frameworks allow the definition of tasks in the form of a declarative language, such as the case of Spark SQL [6]. In these cases, code analysis impacts the performance of the application but it would not be possible to infer anything about the correctness of a program. Other kind of formal methods and analyses could also prove useful towards the goal of improving code quality and their automated nature could provide a mechanism for a continuous evaluation. In particular, symbolic execution [25, 32] is a dynamic program analysis technique that characterizes a program in terms of the boolean path conditions that defines it; one of the uses for such information is the automated generation of unit tests.

1.3 Aim and Research Questions

This study aims to identify if *symbolic execution techniques can be used in the context of Apache Spark as a big data framework to generate reduced input data sets that enforce full path coverage*.

The following research questions are relevant for this study:

1. Is symbolic execution a suitable technique for analyzing programs in the context of Spark applications.
2. What are the particular characteristics associated with the symbolic execution of a Spark program.
3. Is there a symbolic execution framework that can be adapted to perform symbolic executions of Spark programs.
4. If it exists, what are the advantages and disadvantages of such a framework in the context of Spark applications.

1.4 Contributions

This work introduces *JPF-SymSpark*, a symbolic execution framework for Apache Spark programs built as an extension of Java PathFinder (JPF) [57]. The main goal of *JPF-SymSpark* is to generate reduced input datasets that offer full path coverage of the analyzed program. Such datasets can have several uses in the development process of a Spark application, for example, as input data for unit tests.

The tool is able to symbolically execute Spark programs that handle primitive data types and Strings as their input datasets. It is also capable of chaining multiple Spark operations during a symbolic execution;

providing the mechanisms for a complete analysis of a program instead of a method-by-method approach. This reasoning over the interrelation of Spark operations and the data flow among them is the most useful contribution of this work. To our knowledge, there has not been any study over the application of symbolic execution in big data frameworks.

JPF-SymSpark is built on top of Symbolic PathFinder (SPF), a symbolic execution extension of JPF for general Java programs. During the development of the tool, SPF presented unexpected behaviors when analyzing some programs. Most of these abnormal results were caused by common programming practices in Spark applications, for example, the use of anonymous classes and lambda expressions to represent the parameter functions passed to many of Spark's operations. The SPF extension was modified accordingly in order to cope with these scenarios. The modifications introduced to SPF are:

- Detection of synthetic bridge methods
- Consistent ordering of String path conditions
- Improvements to the visitor pattern in the symbolic constraints

Some of these modifications were included in a patch and submitted for revision to the SPF administrator. However, to the date this document was published they have not been included in the official source code. A detailed explanation about the contributions and modifications to SPF can be found in appendix A.

Two evaluations of the tool are presented in this work. The first is a qualitative evaluation that aims to determine how compliant *JPF-SymSpark* is in terms of a series of functional and non-functional requirements defined for a tool of such purpose. The second evaluation consists of a performance assessment of the iterative symbolic operations carried out during an analysis. Additionally, a discussion over the limitations of tool is presented in order to establish the scope and capabilities of its current state.

1.5 Outline

The following chapters are structured in such a way that the reader is able to build the necessary technical knowledge to understand the developed tool properly. Chapter 2 introduces the technologies and concepts on which this work is based. In this chapter, section 2.1 presents the fundamentals of Apache Spark. Section 2.2 briefly explains program analysis as a verification technique and provides an introduction to the concept of symbolic execution. Finally, section 2.3 introduces Java PathFinder and its extension Symbolic PathFinder, a program analysis framework build for the Java programming language and its symbolic execution module respectively.

Chapter 3 focuses on the application of symbolic execution techniques on Apache Spark programs. Section 3.1 analyzes the conceptual implications of using symbolic execution in the context of big data frameworks, section 3.2 presents two concrete examples, and section 3.3 introduces *JPF-SymSpark* and its technical details. It is strongly advised that the reader becomes familiar with the technical concepts, in particular with the intricacies of JPF, before reading this chapter.

Chapter 4 presents the evaluation of the proposed tool and clarifies its limitations. Finally, chapter 5 concludes the work on a discussion over the research questions and suggests a guideline for future work. For further reading, appendix A provides an extended explanation on the contributions done to SPF as a consequence of the work of this thesis, while appendix B presents a guide for the installation and usage of *JPF-SymSpark*.

2 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 in case the concepts explained next are already familiar. Section 2.1 introduces *Apache Spark* as the big data framework under study. Next, section 2.2 presents two program analysis techniques: explicit state model checking, and symbolic execution. Finally, section 2.3 gives an introduction to Java PathFinder (JPF) and its symbolic execution module.

2.1 Apache Spark

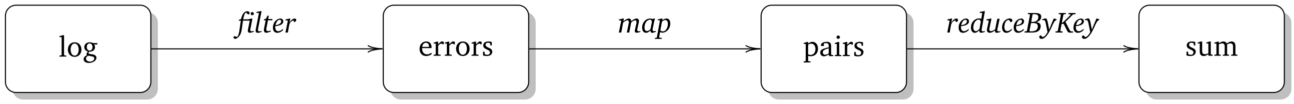
Spark is a distributed data processing framework that was first introduced in 2012 [63]. Similar to other systems, such as MapReduce [15] and Dryad [27], it aims to provide a clean and flexible abstraction to distributed computations on large datasets. However, Spark offers two advantages in comparison 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.

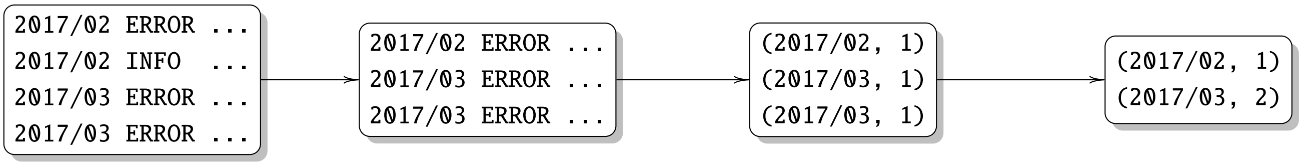
To interact with the RDD abstraction, Spark provides several APIs for different programming languages such as Java, Scala, Python and recently R [55]. Listing 2.1 presents a simple Spark program written with the Scala API, that processes log files in the search for errors. The operation in line 1 creates the first RDD from a log file, whose origin could be a local file or a partitioned file in a distributed file system such 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 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.



(a) Lineage of the program shown in listing 2.1. After each transformation, a new node in the lineage tree is created.



(b) Sample execution of the program shown in listing 2.1. 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 2.1: Lineage and execution of the Spark program shown in listing 2.1. 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) [59]. 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 out. 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 2.1a depicts the resulting lineage of the program explained in listing 2.1 and figure 2.1b 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 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 [60, 4]. Several purpose-specific libraries are built on top of Spark, as is the case of: MLlib for machine learning [38], GraphX for graph computations [61], Spark Streaming for stream processing [64], and Spark SQL, an SQL-like interface for structured querying in Spark [5].

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 [50, 62, 58]. Overall, Spark offers a better performance in comparison to other data processing frameworks.

2.2 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 [42].

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 2.2 and figure 2.2 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 it 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 2.2: 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 -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.

2.2.1 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 grows 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 timely 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 transitions resulting from concurrent operations that lead to equivalent states, making it necessary to explore such paths 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, it results useful for discovering faults that would occur rarely under very specific conditions that cannot be generalized. It is commonly used to validate critical and concurrent systems, and is often combined with other testing techniques.

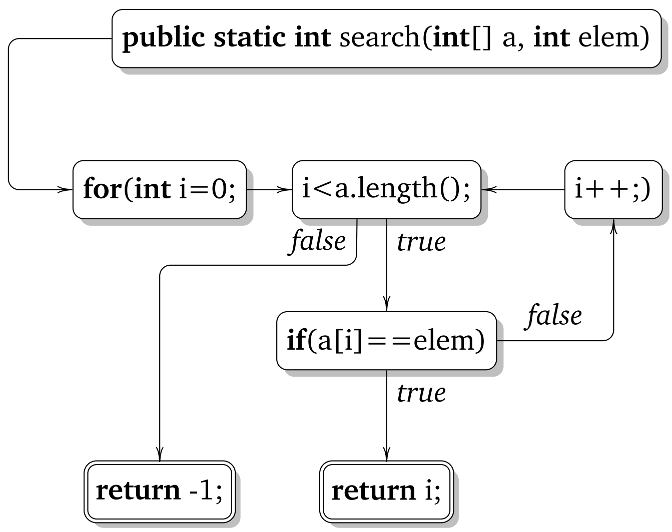


Figure 2.2: Control Flow Graph corresponding to the linear search algorithm shown in listing 2.2. The entry node is the signature of the method, while end nodes, represented with a double frame, contain the return statements that put at end to the execution. The *for* loop instruction was split into its composing statements to better display how the control flow works for this instruction.

2.2.2 Symbolic Execution

The first discussions of symbolic execution date several decades back [25, 32]. 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 unfeasible. Interestingly enough, each satisfiable path condition represents an equivalence class of concrete input values. Figure 2.3 shows the symbolic execution tree of the program in listing 2.3.

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-

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

Listing 2.3: 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 [10].

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 [11], test case generation [9, 12, 20, 56] and static detection of errors [8, 54], among many others [13, 49].

2.3 Java PathFinder

Developed at NASA's Ames Research Center [41], Java PathFinder (JPF) is an execution environment for verification and analysis of Java bytecode programs [57, 30]. Since its publication in the year 2000 [24], 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 [22]. 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 [33].

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

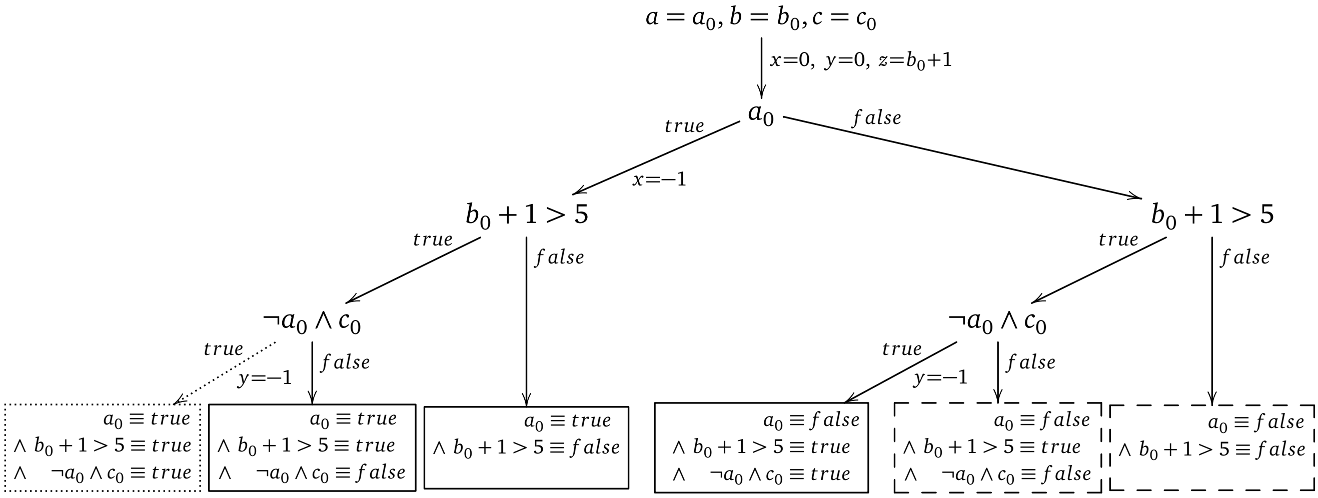


Figure 2.3: Symbolic execution tree of the program presented in listing 2.3. The root node represents the input parameters as symbolic values while the intermediate nodes illustrate the control flow of the program evaluated on these symbolic values wherever possible. Each intermediate node branches into two options, each corresponding to the predicate evaluating to *true* or *false* respectively. Also, each branch is labeled with the executed statements 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 2.4 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, JPF

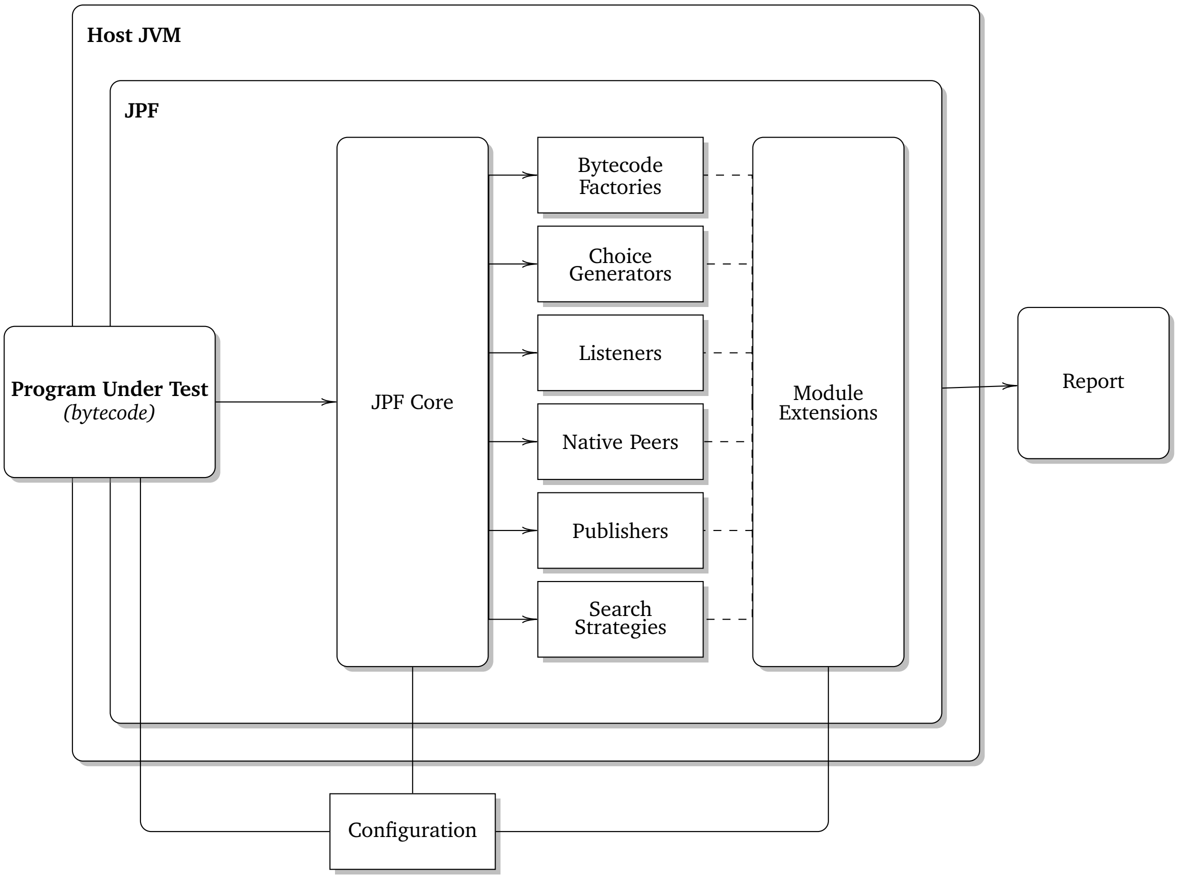


Figure 2.4: JPF Components and workflow. The program under test, taken as bytecode, is loaded into the JPF Virtual Machine which is in turn executed on top of the host JVM. Libraries used in the program under test need to be visible to the core in order to be able to execute the program correctly. Note 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.

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

Listing 2.4 introduces an example that illustrates better how JPF works. The analyzed program 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

```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 2.4: 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 [41])

operation is found promptly and reported back to the user. Figure 2.5 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 the 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 critically reduces 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 successfully, 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

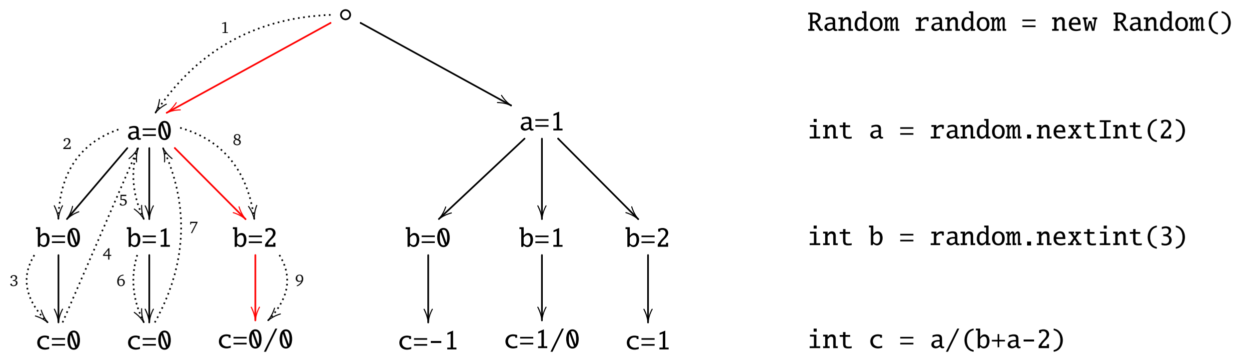


Figure 2.5: State space exploration of the program shown in listing 2.4. JPF starts checking the state space whenever the conditions that trigger the property to be validated are found; in this case, using random values. The `nextInt` instruction causes JPF to register a *Choice Generator* and start 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 the first 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.

2.3.1 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 [31, 45, 2], its current mode of operation consists of replacing the concrete execution semantics of the default JPF model checker with a corresponding symbolic interpretation [44]. In recent years SPF has had some improvements, primarily supporting Java 1.7 and better detection of unfeasible paths [34].

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. Nevertheless, only limited support to symbolic data structures and *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 processes 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 [43], CORAL [51], and CVC3 [7]. 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 [46, 56, 20].

Configuration files are used to indicate which methods should be executed symbolically and also specify which of 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 [19].

3 Symbolic Execution of Spark Programs

To provide a flexible programming paradigm for big data processing, Spark's main API exposes methods that act as higher order functions. Particularly, these methods receive user defined functions as input parameters that dictate how certain operations will be carried out. However, the passed functions always have to comply to some conditions imposed by the method, for example, the function passed to a *filter* transformation must be defined over the data type of the target RDD and must return a *boolean* value.

The use of functions as parameters in Spark operations has an impact on the control flow of the program. Not only the particular Spark operation defines how the program will behave, but also the passed functions could potentially introduce control flow statements like conditionals or loops. Moreover, the control flow behavior of the Spark operations themselves is mostly static (e.g., the iterative and cumulative nature of a *reduce* action), whereas the diverse range of variation introduced by a user defined function is practically unbounded.

For this reason, in the context of program analysis in general and to our particular scope of symbolic execution, both the nature of the Spark operations and the peculiarities of the user defined functions on each program are necessary components that have to be studied together in order to provide a reasonable conclusion.

The following sections describe the conceptual process of a symbolic execution carried out on a Spark program, as well as a detailed explanation of our proposed implementation.

3.1 Conceptual Process

A Spark program consists of a chain of transformations on one or more RDDs that finally conclude with an action applied on them. 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 indicates 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. Many of these operations come from functional programming languages and define abstractions to interact with data collections.

The precise behavior of most operations is defined by the programmer. Given that most of Spark's actions and transformations are higher order functions, 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

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 3.1. 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 instructions that might occur in it.

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 input parameters as symbolic values. 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 operation; 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.

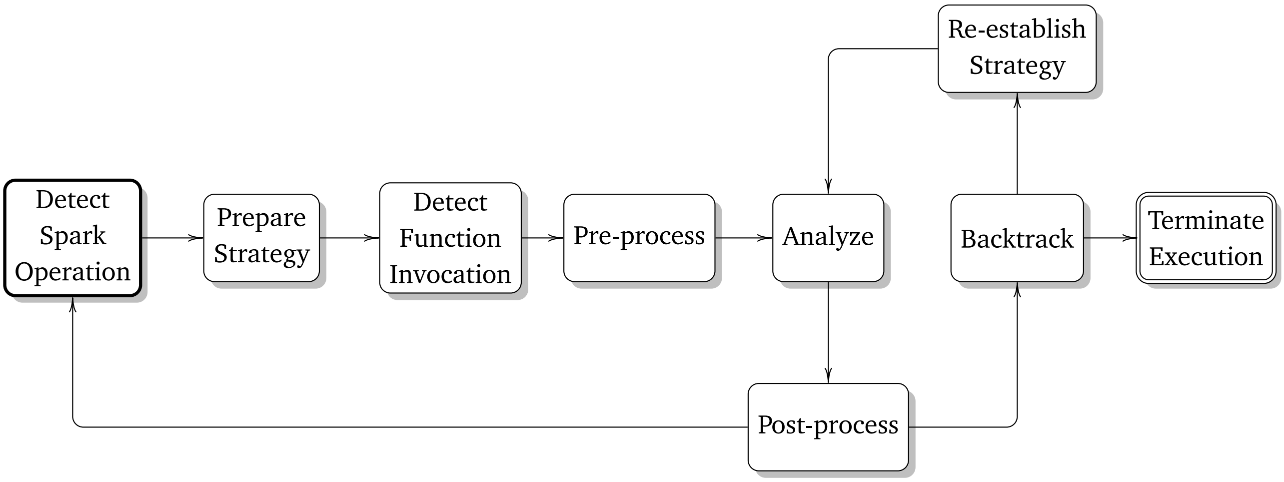


Figure 3.1: State diagram of the symbolic execution process of Spark programs.

3.2 Concrete Examples

Word count [15] has become the de facto example used in studies related to big data frameworks and distributed processing. We present a modified version of the regular word count example that allows us to illustrate the use case of symbolic execution in Spark programs. The goal of the word count algorithm is to process a document or group of documents to determine how many times each word appeared in the analyzed data. The modification we introduce puts a restriction on the structure of the words that are counted; only words that begin with a certain prefix are considered in the algorithm. This modification permits the existence of multiple paths while still keeping the original notion of the word count example.

```
1  JavaRDD<String> textFile = sc.textFile("hdfs://...");
2  JavaPairRDD<String, Integer> counts = textFile
3    .flatMap(s -> Arrays.asList(s.split(" ")).iterator())
4    .filter(x -> x.startsWith("re") || x.startsWith("un")
       || x.startsWith("in"))
5    .mapToPair(word -> new Tuple2<>(word, 1))
6    .reduceByKey((a, b) -> a + b);
7  counts.saveAsTextFile("hdfs://...");
```

Listing 3.1: A modified version of the word count algorithm. Only words that begin with certain prefixes are taken into consideration.

Listing 3.1 shows how the selective word count algorithm is implemented in Spark. The algorithm proceeds as expected; the document is split into composing words and later these words are grouped and counted. The only difference is introduced in line 4, where an additional *filter* transformation is included after the document has been split into words. This operation will filter out all words except those who begin with the “re”, “un”, and “in” prefixes. The only condition imposed over the structure of the input data is introduced by this action, hence it is only relevant to analyze the *filter* transformation in order to determine all possible execution paths.

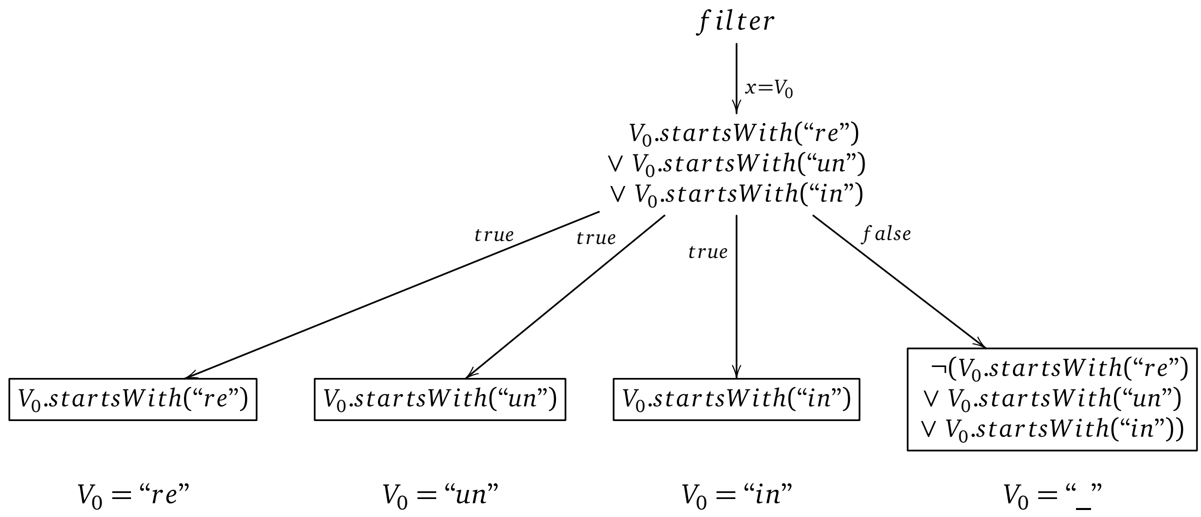


Figure 3.2: Symbolic execution tree corresponding to the Spark program shown in listing 3.1. An input file containing the words "*re*", "*un*", "*in*" and "_" would suffice to explore all feasible paths in the program.

Only two words would suffice to explore all the possible paths of the program; one that would start with the aforementioned prefixes and one that would not. However, in order to try out all the conditions in the disjunction, at least one word matching each prefix would be necessary. Matching each condition in the disjunction is necessary in those cases where the boolean expression defining the path condition has to be thoroughly evaluated. Figure 3.2 shows the symbolic execution tree of the selective word count algorithm. Note that this execution tree depicts an exhaustive exploration of the boolean expression, thus selecting the most simple satisfying word for each condition in the disjunctions (in this case, the prefix itself).

The selective word count algorithm is a good example to convey the usefulness of symbolic execution analyses in the context of big data programs. However, it is not capable of illustrating some particular considerations given the lack of connected constraints between operations.

The trivial example presented in listing 3.2 depicts a simple Spark program with no purpose in itself. Nonetheless, this simple example allows us to better explain how the analysis will be carried out when several relevant operations are present in the program under test. The relevant Spark operations in this example are the *map* and *filter* transformations in lines 4 and 8 respectively. All other operations related to Spark are not relevant.

In this other example, the first operation detected during the analysis is the *map* transformation in line 4. At this point, the analysis opts for a "map" strategy and prepares itself for the imminent invocation of the function passed to the *map*. For convenience, all the functions in this example are depicted as lambda functions although anonymous or named classes would work as well. 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).

During the symbolic execution of the function we find that there is a branching instruction in line 5. This represents a decision point and, for this reason, a *choice generator* is registered with two options:

```
1 List<Integer> numberList = Arrays.asList(1,2,3);
2 JavaRDD<Integer> numbers = spark.parallelize(numberList);
3
4 numbers.map(v1 -> {
5   if(v1 > 1) return v1;
6   else return v1+2;
7 })
8 .filter(v2 -> v2 > 2);
```

Listing 3.2: Trivial example to illustrate the symbolic execution of spark programs with several relevant operations. 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.

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 6. During the post-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.

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 3.3 shows the symbolic execution tree of the program. One interesting aspect to note is that the results of the *map* transformation 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.

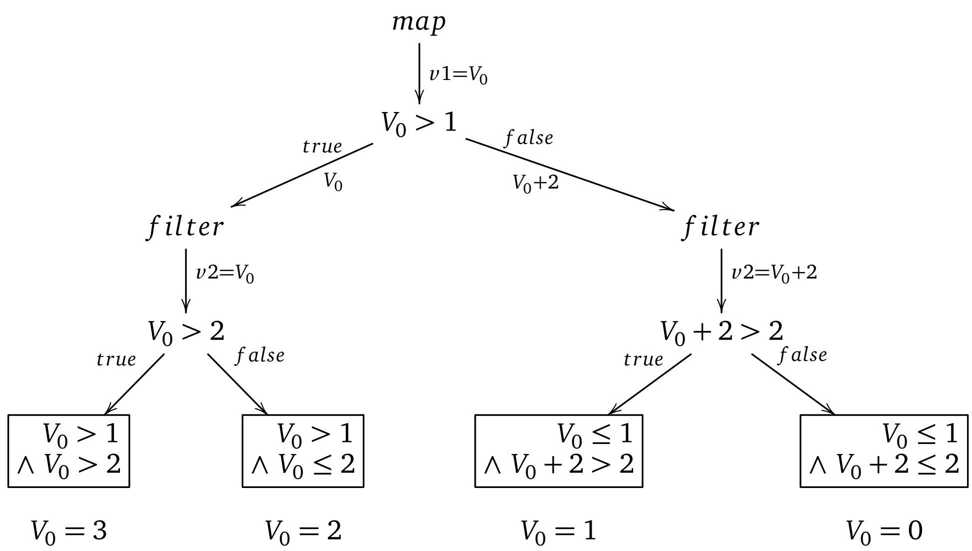


Figure 3.3: Symbolic execution tree corresponding to the Spark program shown in listing 3.2. The input set $\{3, 2, 1, 0\}$ represents a minimal input set that would explore all feasible paths in the program.

3.3 JPF-SymSpark

JPF-SymSpark is a JPF module whose goal is to coordinate the symbolic execution of Apache Spark programs written for its Java API to produce a reduced input dataset that ensures full path coverage on a regular execution. It builds on top of SPF to delegate the handling of symbolic expressions while it focuses on how to interconnect Spark's transformations and actions in order to reason coherently over the execution flow of the program. This section describes the 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 main JPF extension points used in the module are:

- **Bytecode Factory**: Implemented in the *SparkSymbolicInstructionFactory* class, this bytecode instruction factory is in charge of detecting relevant Spark instructions.
- **Listeners**: The main listener is the *SparkMethodListener* class. After the bytecode instruction factory, the listener provides the main interaction point with the analyzed program. In our case, the listener aims to orchestrate the correct symbolic execution of a sequence of Spark operations.
- **Publishers**: The *SparkMethodListener* class also works as a publisher (these two responsibilities are often held together by the same class). The class produces a reduced input dataset obtained after selecting concrete values that satisfy the path conditions.
- **Choice Generators**: In some cases, additional choice generators must be registered in order to

correctly reproduce the behavior of some of the Spark operations. Such is the case of the *SparkMultipleOutputChoiceGenerator* class, which is used when analyzing a *flatMap* transformation, given that it can potentially produce multiple outputs.

Additionally, the module also introduces the following control components that play an important role during the analysis:

- **Validators**: These are abstractions that encapsulate the necessary validations to determine if an executed instruction is a Spark operation or an invocation of the user-defined function. As for the case of the Java platform, the validator is implemented in the *JavaSparkValidator* class.

- **Method Strategies**: Each Spark operation follows a different strategy depending on how they deal with their input and output parameters. One example of a method strategy can be found in the *FilterStrategy* class.

- **Method Coordinator**: The coordinator is in charge of selecting the adequate strategy based on the Spark operation currently being executed. It maintains a general view of the whole analysis. The method coordinator is implemented in the *MethodSequenceCoordinator* class.

- **Surrogate Spark Library**: The regular Spark library undergoes several unnecessary workload that is not relevant to the symbolic execution. Instead of it, a surrogate Spark library was implemented and included as one of the dependencies of the module to relieve the analysis of the irrelevant load.

The following sections explain in more detail the different components that conform the module and what role do they play in the whole analysis.

3.3.1 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 executed program statement, even if it is executed by third-party libraries or dependencies indirectly invoked by 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) because otherwise 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 these libraries and dependencies in the the JPF's classpath and let it 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 operating system; 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 to 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 3.3 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.

```java
import java.util.Arrays;
import java.util.List;
import org.apache.spark.SparkConf;

public class JavaSparkContext {
  public JavaSparkContext(SparkConf conf){}
  public void stop() {}
  public void close() {}
  public <T> JavaRDD<T> parallelize(List<T> list) {
    return new JavaRDD<T>(list);
  }
  public JavaRDD<String> textFile(String file) {
    return new JavaRDD<String>(Arrays.asList(""));
  }
}
```

Listing 3.3: 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.

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, the goal of the surrogate library is that the functions passed to the operations are invoked inside these methods so the analysis can be triggered following the usual SPF approach. Listing 3.4 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.

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

The surrogate Spark library is already included into the dependencies of the *JPF-SymSpark* module and is never meant to be used outside the context of an analysis. Nevertheless, the implementation is not extensive, which might require further expansion as the different cases and programs require. Moreover, the library is bound to version 2.0.2 of the original Spark library, which poses a drawback in terms of

```java
public JavaRDD<T> filter(Function<T,Boolean> f) {
  try {
    f.call(list_t.get(0));
  } catch (Exception e) {
    e.printStackTrace();
  }
  return this;
}
```

Listing 3.4: Mocked filter method in the JavaRDD class. The function passed to the method is invoked. Note that the *Function* interface is also mocked.

consistency if the core behavior of Spark changes in future versions. More about this issue can be found in section 4.3.

3.3.2 Instruction Factory

The first step for carrying out the analysis is to identify when a Spark operation is being executed. Given that all relevant operations in the concrete *JavaRDD* class are implemented as non-static methods, 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) [33].

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 relevant operations. 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 two most common ways lambda expressions 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.

The instruction factory that we implemented, makes use of a validator, which is in charge of detecting the concrete method and class names that are particular for each Spark implementation. At the moment we only support the Java implementation, although the framework is flexible to support additional validators, for example, one that could detect the Scala implementation of Spark.

Moreover, the user must define in a configuration file the method names of the operations of interest (e.g., map, filter) in order to indicate this type of methods should be analyzed. The method names must be defined in a `.jpf` file, in a similar fashion as SPF, under the key `spark.methods`; in the case of having multiple methods of interest, they must be separated with a semicolon.

Once the method has been detected, the instruction factory issues a custom `INVOKEVIRTUAL` instruction to the JPF core. This instruction indicates what should the JPF virtual machine do when the method of interest is executed. In our case, we need to prepare for the subsequent execution of the function passed as a parameter to the Spark operation. To do this, we manipulate the JPF configuration programatically in order to take advantage of the configuration properties used by SPF to define which methods are supposed to be analyzed by the symbolic engine.

Again, let us assume this time that the *filter* method above was executed in a class named `Main` (fully described in `com.test.Main`). Then, after detecting the *filter* transformation, the configuration will contain one of the following entries in the `symbolic.method` key:

`symbolic.method=...;com.test.Main$1.call(sym)` (1)

or

`symbolic.method=...;com.test.Main.lambda$main$0(sym)` (java compiler)
`symbolic.method=...;com.test.Main.lambda$0(sym)` (eclipse compiler) (2)

depending if the passed function was implemented as an anonymous class or a lambda expression.

The "`$1`" in (1) points to the first anonymous class created inside `com.test.Main` (the qualified path for an anonymous class is always separated by a "`$`" sign). The number is monotonically increasing according to how many anonymous classes have been created up to that point. It is important to note that the method *call* indicated here corresponds to the implementation of the method *call* defined in the *Function* interface.

In the case of (2), the method indicated corresponds to a static method defined by the Java compiler whenever a lambda expression is found. This method will invoke afterwards an anonymous class created on the fly which implements the *call* method of the *Function* interface. It is sufficient to indicate the first static method with the symbolic parameter given that it only forwards the execution to the *call* method in the anonymous class. This solution came as a workaround for detecting lambda expressions, given that SPF does not provide any mechanism on its own to clearly specify the analysis of such methods. Same as in the case of anonymous classes, the number accompanying the method is monotonically increasing and depends on how many lambda expressions have been created thus far.

It will be relevant later on, that the method marked to be symbolically executed by (1) will be invoked using `invokevirtual`, while the ones defined by (2) will be invoked using `invokestatic`. This will require the handling of the input and output parameters in a different way.

The dynamic manipulation of the configuration properties makes it easier for the user to specify which methods are to be symbolically executed. Defining the methods before hand, as in the regular SPF approach, proves to be cumbersome, given that the method names corresponding to anonymous classes or lambda expressions are often elusive and even difficult to track.

For further information on how anonymous functions and lambda expressions are compiled and represented in Java bytecode please refer to the Java Language Specification [22].

3.3.3 Spark Listener and Method Sequence Coordinator

The overall process of the *JPF-SymSpark* module is implemented by the *SparkMethodListener* and the *MethodSequenceCoordinator*. The listener acts as a stateless interpreter of the analysis' progress while the coordinator behaves as a stateful control node that provides coherence to the whole process. Both entities work together closely, being the listener just an entry point that dispatches actions to the coordinator based on a selected few relevant events.

Spark Method Listener

The Spark method listener extends the *PropertyListenerAdapter* which already provides entry points to all the events exposed by JPF. The relevant events in our case are:

- **instructionExecuted**: Which is triggered every time an instruction is executed. When this event is triggered, the listener forwards the executed instruction to the coordinator in order to validate if it is a relevant instruction for the analysis.

- **methodExited**: Which is triggered every time a method finishes its execution. In this case, if the method is related to Spark, the listener indicates to the coordinator that it should prepare to percolate the output to possible subsequent Spark methods.

- **stateAdvanced** Which is triggered every time the internal state of JPF advances. This is only relevant to identify that an end state has been reached, in which case the listener indicates to the coordinator that a full path has been explored.

- **stateBacktracked**: Which is triggered every time the execution reached an end state but there were already choice generators registered with unexplored options. The state is always backtracked to the latest point where a choice generator was registered. In this case, the listener instructs the coordinator to obtain a solution to the symbolic input value if the state backtracked to a different alternative of the path condition.

Additionally to these events, the *SparkMethodListener* also implements the *PublisherExtension* interface which allows it to act as a publisher as well. The combination of a listener and a publisher is a common practice among the JPF modules, given it is convenient to collect information worth to be published during the different events tracked by the listener. As a publisher, when the whole analysis is done, the different sample values that satisfied the path conditions that were collected by the coordinator are displayed in the console as a single input dataset. Section 3.3.5 explains in more detail how the output is generated.

Method Sequence Coordinator

The *MethodSequenceCoordinator* class was created as a stateful control structure used to keep track of the progress and results of the analysis. Although it is heavily influenced by the events detected in the listener, the idea was to keep it detached from the event-flow responsibilities of the listener, focusing only on the module's process state.

The goal of the coordinator is to switch to the adequate strategy based on the Spark operations. The different strategies actually do the heavy-lifting in the analysis, and are in charge of handling the behavior of each particular operation correctly. More on the strategies can be found in section 3.3.4.

The main actions of the coordinator are:

- **detectSparkInstruction**: This action determines whether the executed instruction is a Spark operation or the respective invocation if its parameter function. In the case of being a Spark operation, the coordinator selects the adequate strategy matching the operation. Otherwise, if it detects the invocation of the parameter function, then it indicates to the currently active strategy to execute the pre-processing phase.

- **percolateToNextMethod**: In this case, the coordinator instructs the active strategy to execute its post-processing phase. In general, all post-processing activities deal with the inter-connection of Spark operations, this is, dealing with output parameters and the interruption of the control flow, among others.

- **processSolution**: Arguably the most important action of the coordinator. This action is invoked anytime JPF backtracks to a previous state. In the case the state is backtracked to a point where a different path must be taken in a branching condition, that means that the other path has been completely explored. Being this the case, the current path condition is obtained from the choice generator and passed to the solver to determine if it is feasible or not; in the positive case, a sample value of the symbolic input is added to the solution list, otherwise an unfeasible path is reported.

The diagram depicted in 3.4 shows at which point during the execution of the program under test will the listener events be triggered. The *instructionExecuted* event gets triggered on every instruction, however, the depicted points only refer to those instructions that are relevant to the analysis and will cause a respective action in the coordinator; the namely instructions correspond to the invocation of a Spark operation or its respective parameter function. Likewise, the *methodExecuted* event is only relevant when the Spark operation or the parameter function finish. The *stateBacktracked* is indicated to occur at a later

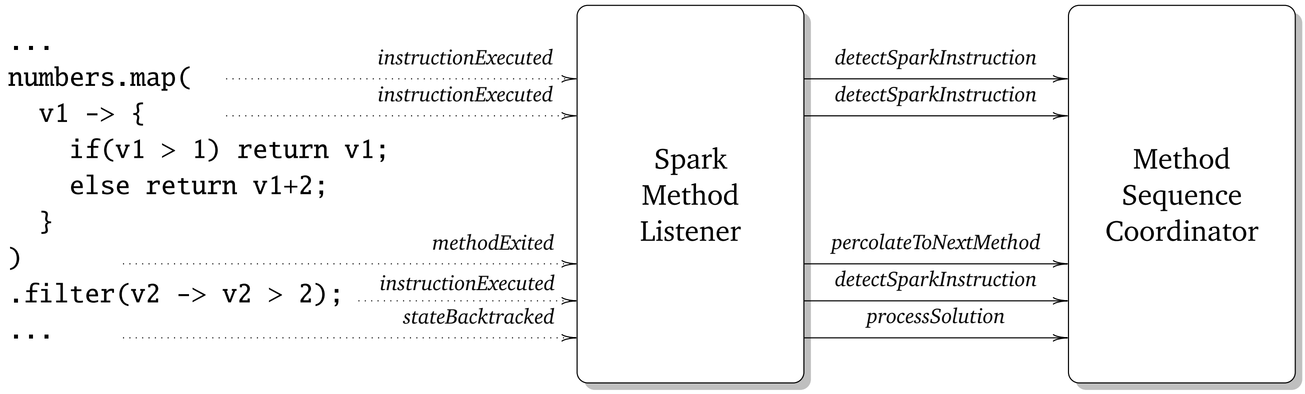


Figure 3.4: Execution flow of the Spark method listener and the coordinator when analyzing the sample program shown in listing 3.2. The exact point where the listener events are triggered actually depends on precise bytecode instructions; this diagram provides an approximation to where this instructions occur in the source code.

point, usually at the end of the program although the backtrack process can be forced without necessarily having reached this point.

3.3.4 Method Strategies

The method strategies specify the concrete behavior of the analysis for each relevant Spark operation. They implement how the analysis is to be carried out, particularly during the pre-processing and post-processing phases. Additionally, they maintain a reference to the input and output values of the operation.

The supported Spark operations are: *filter*, *map*, *reduce* and *flatMap*.

This section explains what particular conditions apply to each strategy and how are they carried out through the analysis.

filter

The purpose of the *filter* transformation is to produce a new RDD containing only the elements that satisfy a given predicate. The predicate is passed to the transformation in the form of a boolean function that is invoked for each element of the RDD. Because of this reason, the *filter* transformation itself always imply at least two possible execution paths, one for those who satisfy the predicate and one for those who do not. Figure 3.5 depicts the symbolic execution of a *filter* transformation according to the filter strategy.

Pre-processing

During the pre-processing phase, the filter strategy only checks for the invocation of the parameter function and validates if it is coming from an `invokevirtual` or an `invokestatic` bytecode instruction. If it is

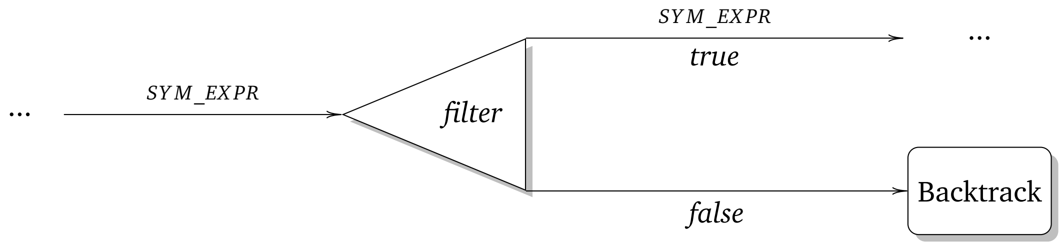


Figure 3.5: Diagram of the *filter* strategy. The input parameter is initialized to the symbolic expression percolated by the previous Spark operation if there is any, otherwise a new symbolic expression is used instead. Filter transformations always produce at least two branches, one that satisfy the predicate and one that does not. In the case of the branch that does not satisfy it, the execution is terminated and an immediate backtrack is triggered. The output of the transformation is always the same input symbolic expression disregarding the path taken.

coming from an `invokevirtual` (the function was implemented as an anonymous class), the strategy manipulates the stack frame of the current invocation and replaces the element in the second position with the symbolic expression passed by the coordinator. This element corresponds to the input parameter of the passed function (i.e., an element of the RDD). The replacement of the second element is necessary because, if this is not done, SPF will call the function with a new symbolic expression instead, breaking the continuity of the analysis. The reason the second element is the one replaced instead of the first is that, in the stack frame of an `invokevirtual` instruction, the first position contains a reference of the invoking object instead (i.e., a reference to `this`). On the contrary, if the invocation of the function comes from a `invokestatic` instruction (the function was implemented as a lambda expression), then the first element is replaced in the stack frame because `invokestatic` instructions do not have references to the invoking object itself.

Given that the *filter* transformation always imply a fork in the control flow, it is most likely that a choice generator was registered by SPF during the execution of the method in order to explore the possible paths. Needless to mention, this only occurs if the symbolic expression passed to the function participates in any of the conditional operations, otherwise, no choice generator is registered; then again, this scenario is irrelevant given that it means that the filter does not act upon the values of the RDD.

Post-processing

On the post-processing phase, the strategy checks if the exited method is actually the *filter* transformation and proceeds to obtain the last registered choice generator. As explained above, this choice generator will always be a path condition choice generator registered by SPF. Then the strategy proceeds to validate if the path currently executing corresponds to the negative evaluation of the predicate, in which case the execution of the thread is abruptly interrupted and a backtrack action is forced. The reason for this relies in the fact that exploring a path with a negative filter condition is not relevant given that this path will never be executed in a Spark program. However, by forcing the backtrack action, the analysis is guided to find a solution that satisfy that path, which ends in a value that does not pass the filter (necessary for full path coverage).

The symbolic input of a *filter* transformation does not suffer any permanent modification during its execution. For this reason, the output of the *filter* transformation is the same input value passed to the function during the pre-processing phase.

map

The *map* transformation constructs a new RDD containing the result of applying the parameter function to each element of the initial RDD. Normally, the input value passed to the parameter function is used in the operation that produces the output value, hence making the output be a derivation of the input value. Considering this rationale in the context of a symbolic execution, the output of the parameter function passed to a *map* transformation is defined in terms of the input symbolic expression. Figure 3.6 depicts the symbolic execution of a *map* transformation according to the map strategy.

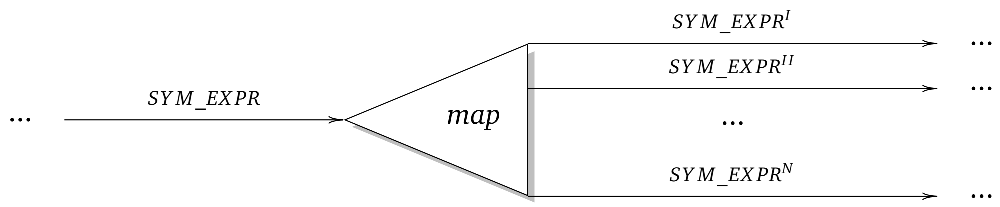


Figure 3.6: Diagram of the *map* strategy. The input parameter is initialized to the symbolic expression percolated by the previous Spark operation if there is any, otherwise a new symbolic expression is used instead. The output value percolated to the following functions is a new symbolic expression derived from any operations applied on the symbolic input. Map transformations do not necessarily have a branching condition, however, in the case there are, it is most likely that the output value will be different.

Pre-processing

The pre-processing phase is identical to the pre-processing phase of the filter strategy. The handling of the input parameters of the passed function is carried out in the same way given that, in both cases, the passed functions have the same number of input parameters, hence their stack frame behaves the same.

Post-processing

In the post-processing phase, the map strategy waits until the passed function finishes its execution. This is done by checking if the exited method matches the full descriptor of the *call* method in either the anonymous class or lambda expression that represents the passed function. Once it detects the right exited method then it stores its output value in an attribute of the strategy. This value will be used the next time strategies are switched and it will be used as the input value of the next Spark operation.

A *map* transformation does not have a branching condition necessarily; it might be the case that the RDD is just manipulated and an output value returned. However, even in this case, the transformation of the symbolic input needs to be tracked in order to percolate it to the next Spark operation. Chaining output and input values between Spark operations is of utmost importance in order to build precise path conditions that faithfully represent the control flow of the program.

In the case the *map* transformation incurs in any branching condition, then SPF will register the respective choice generators and all the options will be explored accordingly. Such a case leads to potentially different outputs depending on which path was taken.

reduce

The *reduce* action produces a single output value resulting from the combination of all the elements in the RDD. The combination is defined in the function passed to action, which has to fulfill the properties of being commutative and associative. The function passed to the action implements the *Function2* interface, whose main difference is that it takes two input parameters: the first is the accumulated value of the operation so far, and the second represents one element in the RDD. The behavior of a *reduce* action resembles to a full scan over the elements of the RDD, carrying always the accumulated value iteration over iteration.

Reduce actions can be analyzed following two different strategies: one where the accumulated parameter of the function is not considered as a symbolic variable and another where it is. The strategy discussed next refers to the former; the latter is explained in the subsequent section. The user indicates which mode to use by specifying it in the configuration file.

Pre-processing

This strategy considers the input parameter corresponding to a single element in the RDD as the percolated symbolic expression, however, the parameter corresponding to the accumulated value of the reduce action is considered as a concrete input.

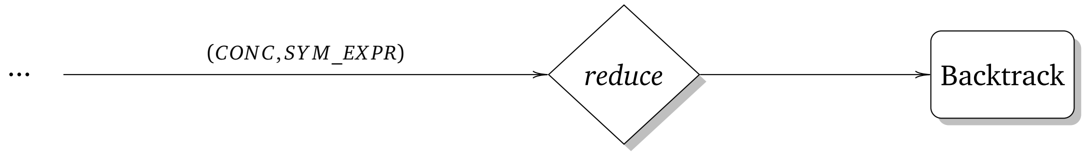
The analysis takes advantage of the concolic operational mode of SPF by defining the method to be inspected with the first parameter as a concrete input (achieved by using the “con” keyword instead of “sym” in the fully qualified method name). Afterwards, in a similar fashion as in the filter and map strategies, the second or third element in the stack frame is replaced with the percolated symbolic expression (depending if the instruction is `invokevirtual` or `invokestatic`). Figure 3.7a illustrates this mode.

Post-processing

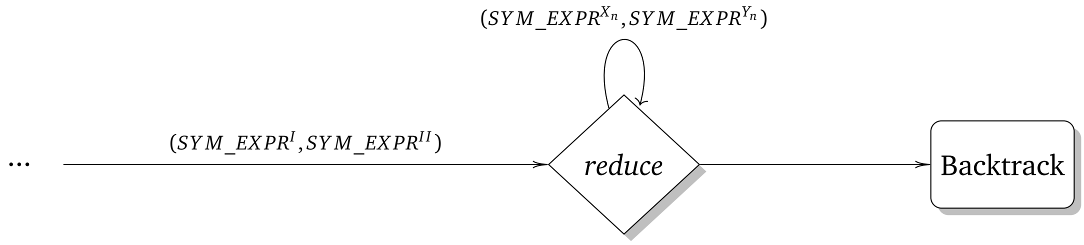
Independently of the output of the *reduce* action, the post-processing phase of the reduce strategy always triggers a termination of the current execution thread and a backtrack. This is because of the nature of Spark actions which indicate the culmination of the processing of an RDD. However, during the analysis of the reduce action, there might be branching operations executed on the single parameter, which will cause SPF to register a choice generator and explore all the possible paths. Ultimately, all these paths will lead to a backtrack after the end of the *reduce* action.

iterativeReduce

The second strategy that can be used when analyzing *reduce* actions considers both parameters of the passed function as symbolic. The direct consequence of this consideration is that the symbolic engine has



(a) Reduce strategy with the accumulated value taken as a concrete input.



(b) Reduce strategy with the accumulated value taken as a symbolic input. The value n represents a given iteration.

Figure 3.7: Diagrams of the *reduce* Strategy. The two modes of execution are shown here. The first considers the accumulated input parameter as concrete, this translates to only considering branching operations applied on single elements of the RDD. The other mode considers the accumulated to be a symbolic input as well and iterates over the operation a fixed number of times. A backtrack is always triggered after a reduce action.

to reason now over a variable that represents an accumulated value resulting from the application of the passed function several times. Given that the numbers of elements in the RDD is irrelevant for the sake of the analysis, the user has to specify how many iterations of the *reduce* action will be carried out in oder to ensure termination. However, path explosion can occur easily given that each iteration makes the number of reachable states grow exponentially.

The complexity of this strategy is noticeable in comparison to the other strategies. The expected output dataset now becomes a family of datasets corresponding to the number of elements indicated to be in the RDD. The path conditions take into consideration multiple symbolic variables that have to comply with all the transformations and constraints collected so far. The particular aspects of the output datasets are detailed in section 3.3.5.

Pre-processing

As mentioned before, this strategy considers both parameters of the function passed to the *reduce* action as symbolic variables. Once the *reduce* actions is detected, a new choice generator called *SparkIterative-ChoiceGenerator* is registered. This choice generator is used to keep a reference of the number of times the function has to be executed and it also keeps track of the output of each of the iterations; it is not used as a value provider in a direct sense rather as a placeholder to keep track of the iterative execution. The current path condition, if any, is also kept in the choice generator along with the single symbolic variable used so far. This value will serve as a template for creating new symbolic variables that comply with the conditions accumulated to this point.

Once the execution of the *call* method is detected, the strategy follows one of two approaches: If there are no accumulated values registered in the *SparkIterativeChoiceGenerator* choice generator then a new symbolic expression is created taking into consideration the base structure of the percolated expression. This new expression is used as the first accumulated value while the percolated expression is used as the single input value. On the other case, an output value (representing the result of a previous iteration) is taken from the choice generator and set as the accumulated value while a new symbolic expression is produced and set as the regular single input value of the function. Figure 3.7b illustrates this mode.

As a technical note, symbolic expressions and path conditions are implemented in SPF following the *visitor* design pattern [17]. New visitors were implemented in order to create copies of existing expressions and conditions among others.

Post-processing

Once the *call* method finishes execution the current path condition is extended by enforcing the same initial constraints on the newly generated expressions for this iteration. Additionally, any constraints accumulated from a previous iteration must also be included in the current path condition. This has to be done at this point because JPF is not really executing an iteration, instead the iteration is simulated with the *SparkIterativeChoiceGenerator*, which will cause the path conditions to be solved after the method finishes and the engine is set to backtrack to the latest point (most likely the point where the *SparkIterativeChoiceGenerator* was registered), hence missing interconnection between the iterations. Including this conditions ensure that the right path in the symbolic execution tree is taken.

Lastly, the output expression of the method is stored in the the choice generator along with the current path condition. This will serve as an input for a subsequent iteration in case the maximum number of iterations has not been reached yet.

flatMap

The *flatMap* transformation behaves similarly to the *map* action, the only difference is that the function passed to the *flatMap* has a collection of elements as an output value instead of a single element. These elements could have undergone different transformations even though they are returned in the same collection. One example of this behavior can be seen in listing 3.5. Here, two different iterable collections are returned, one contains two elements resulting from two different manipulations of the initial input, while the second contains only one element with another manipulation different from the other two. Figure 3.8 depicts the symbolic execution of a *flatMap* transformation according to the flatMap strategy; it shows how each path taken inside the passed function ends up in the registration of a new choice generator whose options are the possible symbolic expressions in the returned collection.

Pre-processing

The pre-processing phase is identical to the pre-processing phase of the filter and map strategies. Again, the handling of the input parameters of the passed function is carried out in the same way because the passed functions have the same number of input parameters, hence their stack frame behaves the same. However, the function passed to a *flatMap* transformation implements the `FlatMapFunction` interface,

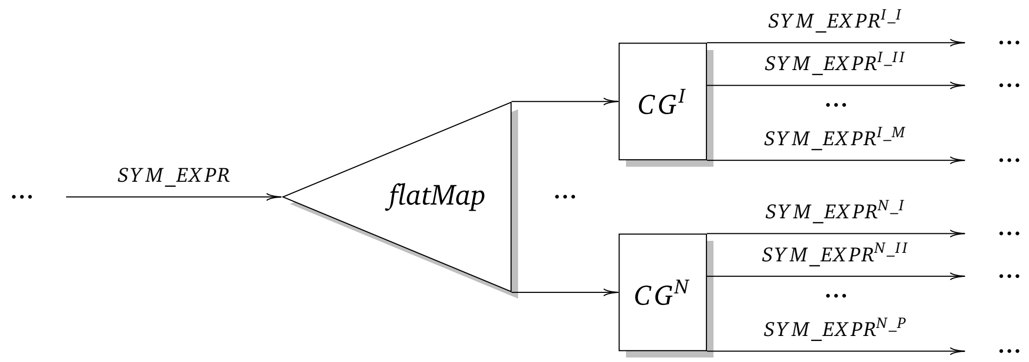


Figure 3.8: Diagram of the *flatMap* strategy. The input parameter is initialized to the symbolic expression percolated by the previous Spark operation if there is any, otherwise a new symbolic expression is used instead. The output value percolated to the following functions is taken from a special choice generator registered when the *flatMap* transformation has finished executing. The choice generator contains all the different elements contained in the collection. Every path taken during the *flatMap* transformation registers a new choice generator with all possible values returned in that particular path.

```
numbers.flatMap(t -> {
if(t > 2) return Arrays.asList(t*2, t*3).iterator();
else return Arrays.asList(t*4).iterator();
});
```

Listing 3.5: A trivial example of the *flatMap* transformation. It shows how the returned iterable can have elements that have undergone different transformations.

which differs from the `Function` interface used by the other two transformation by returning an the iterator of a collection instead.

Post-processing

The post-processing phase of this strategy behaves quite differently from all the other strategies so far. The idea would be to obtain all the different symbolic expressions inside the iterable object returned by the function and use each of them to feed subsequent Spark operations. However, the problem here lies in the change of the cardinality of the possible outcomes; so far we have had always the scenario of one input producing exactly one output, but now one input could potentially imply several output.

For this purpose, the strategy relies on how the mocked up *flatMap* transformation is implemented in `JavaRDD` class. In this implementation, the returned iterable is completely traversed using the *next* method, forcing the execution to bring each element of the collection into the stack frame of the *flatMap* method. This implementation does not deviate starkly from the original *flatMap* operation given that in the case of the regular Spark library, the whole iterator is explored to build a new RDD.

Considering how the implementation behaves, the post-processing phase waits until the passed function exits and then checks for an `invokevirtual` instruction invoking the *next* method. Once detected, the respective value is taken from the stack frame and added to a list of output values in the strategy. This list is filled up with every element in the iterator.

Lastly, when the *flatMap* transformation exits, the strategy register a new custom-made choice generator whose possible values are the collected output values. Registering a choice generator a this point ensures that, after backtracking, a new option from the collection will be chosen exactly at the end of the *flatMap* transformation. The selected value will be used as input of any subsequent Spark operations and the analysis will continue accordingly.

The choice generator used in this scenario is called `SparkMultipleOutputChoiceGenerator`; it extends the `IntIntervalGenerator` of the JPF core. It contains a list of symbolic expressions representing the different manipulations of the input value and uses the integer range to select one of this options every time a backtrack occurs. The range is defined between zero and the size of the list minus one.

3.3.5 Output

After a symbolic execution, *JPF-SymSpark* produces reduced input datasets that ensure the full path coverage of the program under test. The elements in these datasets are collected during the execution and are presented to the users through the implementation of the *PublisherExtension* interface provided by JPF. By the means of the publisher interface, the datasets are included as part of the execution summary of JPF and subsequently printed in the standard output. There are two types of datasets that can be generated as an output of the analysis depending on which strategy was chosen: the regular dataset and the iterative datasets.

Regular dataset

A regular dataset contains one representative element of each equivalence classes defined by the satisfiable path conditions found during the symbolic execution of the program under test. This dataset is produced when the program under test does not contain a *reduce* action or the analysis was not set to conduct an iterative symbolic execution of the said operation. In such a scenario, a regular dataset is sufficient to ensure full path coverage of the program under test. The reason for this is that because no iterative analysis is required, the interrelation between the elements of the input dataset is irrelevant.

This kind of output is sufficient when the goal of the analysis is to reason over a series of transformations. By definition, transformations in Spark have a one-to-one or one-to-many semantics (for example, *filter* and *flatMap* respectively) but never a many-to-one semantics as in the case of the aggregation actions.

Iterative datasets

On the contrary, iterative datasets are produced when the analysis is set to conduct symbolic executions of iterative actions. In this case, the interrelation of the elements in the dataset is relevant to possible path conditions defined over accumulated values. For this reason, we decided to produce a dataset for each path condition found during the execution of the program under test, where each element of the dataset

```
Current path condition not satisfiable: constraint 3
v1_4_SYMINT > CONST_2 &&
(v1_4_SYMINT + CONST_1) <= CONST_3 &&
v1_1_SYMINT > CONST_2
```

Listing 3.6: Output for an unsatisfiable path condition. Values containing the word `SYMINT` respresent symbolic integers while values of the form `CONST_X` are plain integer constants representing the number replacing the `X`. This path condition has a contradiction between the first and second elements of the conjunction.

corresponds to a symbolic input that participated in the cumulative action. The cardinality of the dataset increases based on the number of iterations given that more iterations need larger datasets to actually be executed.

The result is presented as a family of datasets that comply to all the path conditions found during the execution. We decided to present all datasets and not only those corresponding to the final iteration because, although datasets of smaller iterations incur in some redundancy, it might be the case that the final iteration incurs in an unsatisfiable path condition for a path that was feasible in a previous iteration. Determining when a dataset is redundant is a future optimization.

Furthermore, an analysis can have both a regular dataset and iterative datasets as output. Although a program under test is analyzed following an iterative strategy, some transformations can break the control flow before reaching the aggregate action. This is the case of *filter* transformation, where the negative branch is immediately backtracked after execution. In this scenario, a representative element for all possible path conditions that came to an end before reaching the iterative action is included in a regular dataset.

Unsatisfiable path conditions

Additionally, *JPF-SymSpark* also reports those path conditions that were not satisfiable. This process gets triggered after a backtracking action resulting from the complete exploration of an execution path. If the path condition defining the explored path is satisfiable then a result is included to the respective dataset, otherwise the unsatisfiable path condition is reported directly to the standard output which in many cases is the executing console. Listing 3.6 shows the output corresponding to an unsatisfiable path condition of a sample program.

The notification of unsatisfiable path conditions allows the users to identify control flow operations that could potentially represent incorrect or unexpected behavior. Altogether with the reduced input datasets, the notification of unsatisfiable path conditions provides the remaining missing part for a comprehensive analysis on the symbolic execution of a Spark program.

4 Evaluation

This chapter presents the evaluation of *JPF-SymSpark* in two dimensions: a qualitative examination of the overall tool and a quantitative appraisal of iterative symbolic executions. Additionally, the chapter concludes with a discussion about the limitations of the tool and its processes.

The qualitative examination aims to contrast *JPF-SymSpark* against a series of functional requirements of an ideal symbolic execution framework for Apache Spark. The quantitative appraisal of the iterative symbolic executions explores the behavior of the iterative reduce strategy in order to highlight performance obstacles when choosing this execution approach. Lastly, the discussion on the limitations provides additional information on the constraints, for both the tool and the process, under the context of JPF.

4.1 Qualitative Evaluation

For the qualitative examination, we specify a series of conceptual requirements that define the functionality of an ideal tool used to conduct symbolic executions on Spark programs. *JPF-SymSpark* is then compared against these requirements in order to determine how successful the implementation of the module is. Given that, to our knowledge, *JPF-SymSpark* is the only tool capable of carrying out symbolic executions on a big data framework, it is relevant to identify what are the main requirements for a tool with such a goal in order to define a complying criteria set for further evaluations.

The requirements presented in this section are as generic as possible, without falling into the specifics of any architecture or programming language. They focus on the capacity of generating artificial input values that ensure full path coverage, which is the ultimate goal of this research work. The following list presents the nine identified requirements and also provides a brief explanation for each of them.

4.1.1 Requirements

R.1 *The framework should produce a reduced input dataset that ensures full path coverage of the program under test*

This is the core requirement for such a framework. The expected output of a symbolic execution in this case should be a dataset with as few elements as possible that can be used as input in a regular run of the program under test to ensure full path coverage. There could be several use cases for such an input dataset, for example, the generation of automated unit tests that assert the correct termination of the program.

R.2 *The framework should report unfeasible path conditions in case there are any*

Unfeasible paths are a sign of faulty implementations or wrong design assumptions. It is highly desirable that the framework notifies when an unfeasible path is found because such information will aid developers to focus on flawed portions of the source code.

R.3 *The framework shall conduct a symbolic execution of all the operations that conform the program, ensuring the correct interconnection between consecutive transformations and actions*

A holistic analysis of a Spark program is only reasonable if intra-procedural and inter-procedural evaluation are ensured. Conducting symbolic executions on each relevant operation independently is not sufficient to argue on the whole data flow. For this reason, ensuring the right propagation of symbolic values among operations is crucial for a significant analysis.

R.4 *The framework shall conduct a symbolic execution of the program under test without requiring any modification to its source code*

Black-box approaches promote the adoption of the analysis tool and ensure that the program under test is faithfully a representation of a real program. Additionally, such approaches could be integrated with developer environments in order to conduct automated analyses and provide suggestions in real-time, whereas this could not be possible if the tool would require the manipulation of the source code.

R.5 *The framework shall be able to reason over symbolic primitive types*

Primitive types represented as their respective wrappers classes in Java should be supported. Symbolic path conditions for these types are often represented as linear and non-linear arithmetical equations. However, few Spark programs work solely on RDDs of simple primitive types, more elaborated data structures are used often.

R.6 *The framework shall be able to reason over symbolic Strings*

Spark is frequently used to process large amounts of text input, usually in the form of whole files. One example of such an algorithm is the calculation of an inverted index, used commonly in search engines to map key terms to the document where they are found. For this reason, supporting symbolic String operations is fundamental for such a framework.

R.7 *The framework shall be able to reason over symbolic data structures*

In a similar form, many Spark programs work on more complex data structures. Tuples in particular are widely used, having even in some cases specific operations defined on them, such is the case of *reduceByKey*.

R.8 *The framework should support all Spark programs that compile correctly*

A valid Spark program should always be supported even if there is no relevant dataset to be inferred. This enforces the usability of the framework.

R.9 *The framework shall be able to process iterative and cumulative actions*

Some Spark actions, such as *reduce*, have an iterative behavior over the elements of an RDD. In some cases an accumulated value could participate in branching operations; this causes the path conditions to change for every element processed. The framework should be able to reason about how this path conditions will change after every iteration and provide different input datasets that explore their respective paths based on how many elements they contained.

These general requirements are sufficient to describe a tool for the symbolic execution of Spark program by dealing with the specific conditions of the big data framework. Further studies can use and extend this primary requirements to conduct additional evaluations to other data intensive frameworks.

4.1.2 Validation

For the validation, each requirement introduced in section 4.1.1 is revisited and discussed in the context of *JPF-SymSpark*. We indicate first whether *JPF-SymSpark* complies or not with the requirement and, subsequently, provide an explanation on how this was concluded. On occasions requirements are only partially met, in which case the reasons for a partial fulfillment are explained. However, section 4.3 offers a thorough discussion on the limitations of the tool. Moreover, table 4.1 presents a summary of the validation results.

R.1 *The framework should produce a reduced input dataset that ensures full path coverage of the program under test*

This requirement is fulfilled. As explained in section 3.3.5, *JPF-SymSpark* produces two types of output as a consequence of the symbolic execution. The first is a single input dataset containing a value for each satisfiable path condition found. This dataset is in fact minimal given that only one value per path condition is taken. The second type is a family of input datasets produced as a consequence of the symbolic execution of iterative aggregate operations. In this case each element in the dataset follows a single path condition, however, its cardinality indicates the number of iterations considered in the analysis.

Certifying that the generated datasets actually ensured full path coverage comes as a direct consequence on how the elements of the dataset were obtained. However, we used *JaCoCo* [29], a code coverage analysis tool for Java programs that is capable of measuring different levels of coverage in order to verify the assumption in several examples. Although there is no tool for the Java programming language that is capable of measuring path coverage, *JaCoCo* is able to measure branch coverage and cyclomatic complexity [37] (only intra-procedural), which gives an approximation to path coverage metrics. In all the evaluated examples, the obtained datasets offered full coverage in the two aforementioned metrics.

R.2 *The framework should report unfeasible path conditions in case there are any*

This requirement is fulfilled. Again, as explained in section 3.3.5, *JPF-SymSpark* reports unfeasible path conditions as soon as they are found.

R.3 *The framework shall conduct a symbolic execution of all the operations that conform the program, ensuring the correct interconnection between consecutive transformations and actions*

This requirement is fulfilled. The main contribution of *JPF-SymSpark* is to provide a framework that allows the symbolic execution of consecutive Spark operations by correctly connecting the respective input and output values of each operation. This approach allows a comprehensive analysis of a program instead of a method-by-method reasoning.

R.4 *The framework shall conduct a symbolic execution of the program under test without requiring any modification to its source code*

This requirement is fulfilled. *JPF-SymSpark* was designed to work as a black-box tool. The surrogate Spark library is only included in the JPF's classpath as it is specified in the corresponding *.properties* file, making it only available during the analyses. Normal executions of the program under test do not include the surrogate library, using instead the official Spark library.

R.5 *The framework shall be able to reason over symbolic primitive types*

This requirement is fulfilled. Spark operations over RDDs of *Integer*, *Long*, *Float*, *Double* and *Boolean* wrapper classes are supported.

R.6 *The framework shall be able to reason over symbolic Strings*

This requirements is partially fulfilled. Support on symbolic String operations is constrained by the limitations of SPF. This poses a major limitation for the adoption of the tool given that many big data tasks rely on text processing. More on this in section 4.3.

R.7 *The framework shall be able to reason over symbolic data structures*

This requirement is not fulfilled. Support to symbolic data structures in SPF is faulty; as a consequence, *JPF-SymSpark* is not capable of reasoning on RDDs of any complex data structures. This represents a major limitation given that Tuples are frequently used in big data tasks to group data.

R.8 *The framework should support all Spark programs that compile correctly*

This requirements is partially fulfilled. *JPF-SymSpark* relies on a surrogate Apache Spark library that is used for two purposes: First, to relief JPF from processing irrelevant operations for a symbolic execution, such as context initialization. The second, simplify the methods in the RDD's API in order to facilitate the symbolic executions, for example, removing loops and additional considerations relative to distributed computing. This library is not exhaustive, for this reason there will be unsupported operations that compile under the regular Spark library. Furthermore, other Spark APIs, such as the Dataset API, are not supported.

	R.1	R.2	R.3	R.4	R.5	R.6	R.7	R.8	R.9
JPF-SymSpark	✓	✓	✓	✓	✓	†	×	†	†

Fulfilled ✓ Not fulfilled × Partially fulfilled †

Table 4.1: Summary of the qualitative validation of *JPF-SymSpark*.

R.9 *The framework shall be able to process iterative and cumulative actions*

This requirement is partially fulfilled. Only actions that work on primitive values are supported. As a consequence of *R.7*, all the actions that work on symbolic data structures cannot be processed. Moreover, the symbolic output of aggregate functions is not percolated beyond the boundaries of the operation; this means that any branching condition applied on the return value of an action is not symbolically executed.

Although most of the requirements are met, those that are only partially met or not met at all represent severe obstacles to the applicability of the tool on less trivial Spark programs. Nevertheless, the proposed process and the foundation of *JPF-SymSpark* lay the ground for further research on this topic.

4.2 Quantitative Evaluation of Iterative Symbolic Executions

The process carried out by the iterative reduce strategy resembles to a loop unwinding technique, where an iterative behavior is taken out of a loop and executed sequentially without the need of accumulators or counters. This approach is often used to improve execution speed of a program by getting rid of all the control flow instructions inherent to loop statements. However, regarding *JPF-SymSpark* and several other program analysis techniques, loop unwinding is used to bound the execution of potentially infinite loop instructions, hence allowing the evaluation of the program under test.

JPF-SymSpark allows the user to specify the number of iterations of an aggregated function that will be symbolically executed. This enables the module to bound the execution of the iterative behavior and carry out the analysis by correctly chaining the outcome of previous iterations. However, the number of path conditions grow exponentially with the number of conditional statements found. This means that if the aggregate function being analyzed has a conditional statement, then it is most likely that a path explosion will occur.

This evaluation focuses on the behavior of the iterative reduce strategy and it aims to identify the major aspects that pose performance losses. It illustrates how an increment in the number of iterations to be considered in the analysis has a direct impact in the number and size of path conditions, as well as in the performance of the constraint solvers. Moreover, it serves as an example to identify path explosion in symbolic executions.

4.2.1 Setup

Four scenarios were considered for the experiments:

- **Iterative Reduce with Non Cumulative Condition (IRNC)**: The program under test contains a single *reduce* action with a conditional instruction defined only over a non cumulative symbolic variable.

- **Iterative Reduce with Cumulative Condition (IRC)**: Also a single *reduce* action but this time the conditional instruction is defined over the cumulative symbolic variable.

- **Iterative Map and Reduce with Non Cumulative Condition (IMRNC)**: This scenario is similar to the first one with the difference that the *reduce* action is preceded by a *map* transformation with a single conditional statement over its symbolical variable.

- **Iterative Map and Reduce with Cumulative Condition (IMRC)**: Same as the previous scenario except that the *reduce* action has a conditional instruction defined over its cumulative symbolic variable.

The first two scenarios aim to determine what kind of performance effects generate from solving constraints based on cumulative and non cumulative constraints. The last two scenarios are used to measure the relevance of previous transformations that manipulate the symbolic variable and also introduce a condition to the path before the `reduce` action. All the scenarios were implemented as trivial Spark programs processing only integer values. Furthermore, the conditional statements between operations were defined in a way that unsatisfiable path conditions would be generated eventually.

Each scenario was executed for 2, 3, 5, 8 and 13 iterations respectively, repeating each case ten times and taking the average of the execution time as the result. This range follows a Fibonacci sequence and was chosen to space out the iterations enough to make any trend distinguishable. Additionally, for each case, the number of satisfiable and unsatisfiable path conditions was registered. The experiments were carried out in a laptop computer with an Intel Core-i7-6500U processor at 2.50GHz and assigning 1GB of memory to the JVM executing the analysis. JPF was triggered using the command line instead of the Eclipse JPF plugin in order to avoid wasting memory in processes inherent to the Eclipse IDE.

4.2.2 Results and Discussion

The results of the experiments are summarized in table 4.2 for the executions times, and in table 4.3 for the number of satisfiable and unsatisfiable path condition for each scenario.

Execution times

All the scenarios have a similar performance up until three symbolic iterations. However, the performance of some scenarios start to diverge after five iterations. The starkest difference is displayed by the *IRC*

Iterations	2	3	5	8	13
IRNC	0.680	0.705	0.855	1.344	5.489
IRC	0.589	0.625	15.616	568.004	N/A
IMRNC	0.669	0.710	0.981	2.621	21.662
IMRC	0.681	0.728	1.144	2.585	177.579

Table 4.2: Average execution time in seconds for each scenario under the different number of executed iterations.

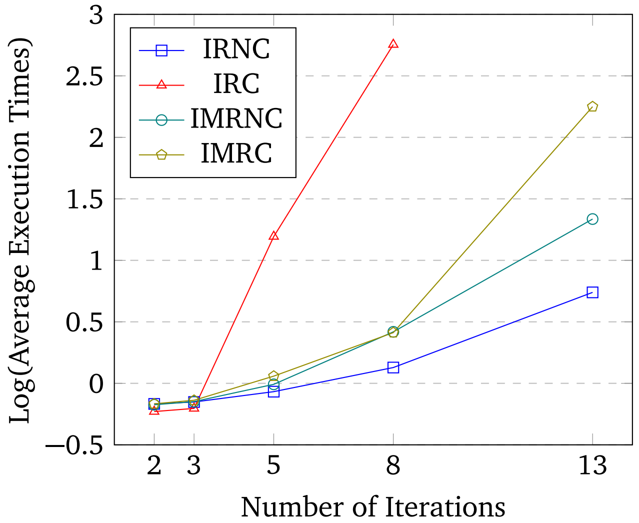


Figure 4.1: Average execution times in logarithmic scale. The disparity between the scenarios is easily perceived using a logarithmic scale while still preserving the notion of exponential growth.

scenario, where the execution times increase several orders of magnitude in comparison to the other scenarios. In particular, the *IRC* scenario could not be executed for 13 iterations or more, resulting in executions that lasted several hours and culminating eventually in *OutOfMemory* exceptions.

The rest of the scenarios have a steady increase in their measurements until they reach 13 symbolic iterations, where the execution times start to grow exponentially. From the remaining three scenarios, the *IMRC* scenario displays the worst performance, taking almost three minutes to execute 13 symbolic executions. Figure 4.1 presents a plot of the execution times in a logarithmic scale that better depicts the disparity between some of the scenarios.

Number of path conditions

As expected, the total number of path conditions grows exponentially over the number of iterations executed. The scenarios having a *map* transformation before the iterative action have double the total amount of path conditions than their counterparts without the transformation. This is consistent to the exponential growth of the number of path conditions given that the transformation only introduced an additional branching statement.

Nevertheless, the number of non satisfiable path conditions varies strongly depending on the scenario

Iterations	2		3		5		8		13	
	s	u	s	u	s	u	s	u	s	u
IRNC	6	0	14	0	62	0	510	0	16382	0
IRC	6	0	14	0	62	0	454	56	n/a	n/a
IMRNC	8	4	17	11	67	57	518	512	16395	16369
IMRC	7	5	12	16	25	99	57	963	573	32191

Table 4.3: Number of satisfiable and unsatisfiable path conditions for each scenario under the different number of executed iterations. As a reference, the letter "s" stands for satisfiable while the letter "u" stands for unsatisfiable.

being studied. In the case of the *IRNC* scenario, all path conditions are satisfiable, while for the *IRC* scenario, only a few in the case of 8 iterations are not satisfiable; most of these resulting from timeouts of the constraint solvers. On the contrary, the *IMRNC* and *IMRC* scenarios present high rates of unsatisfiable path conditions throughout the different number of iterations, ranging from 30% to 50% for the former and from 45% to 90% for the latter.

Discussion

The results obtained in these experiments highlight some key features on the behavior of symbolic execution of aggregate functions. First, the performance of the framework is considerably better when the conditional statements defined in the aggregate function only affect the non-cumulative symbolic variable in the operation. This comes as a consequence of the kind of constraints generated in such cases. In the case of non-cumulative conditions, the constraints are defined over single independent symbolic variables that are joined in a conjunction on each iteration, while in the case of cumulative conditions, after each iteration, the constraints include more symbolic variables related with each other. Constraint solvers perform better when the constraints are defined over independent symbolic variables, which explains the behavior of the *IRC* scenario and the difference between the cumulative and non-cumulative scenarios in general.

Another aspect is the difference between the *IRC* and *IMRC* scenarios. After five iterations, the behavior of these two scenarios diverges strongly, having the *IRC* scenario performing exponentially worse and even reaching the point of not being able to finish its execution, while the *IMRC* scenario is still capable to finish the execution within a reasonable amount of time. The reason for this relies in the number of unsatisfiable path conditions in the *IMRC* scenario because, although this scenario has more path conditions than the *IMC* scenario, a large number of them are unsatisfiable, allowing the constraint solver to conclude faster. This result indicates that having more path conditions does not necessarily translates into worse performance; it depends on how complex the path conditions are and how many of them are actually satisfiable.

The performance of the constraint solver as well as the complexity of the path conditions are the major factors that affect the overall performance of iterative symbolic executions in *JPF-SymSpark*. Moreover, the construction of aggregate functions with conditional statements has to be properly evaluated in terms of the congruency of the operation. Conditions applied to a cumulative value can break the associativity requirement of reducer functions, leading to invalid, non-parallelizable Spark programs.

4.3 Limitations

This section comments on several limitations of *JPF-SymSpark* that come as a direct consequence of the design decisions or are caused by any of the frameworks upon which the module builds up.

The processing logic of *JPF-SymSpark* focuses on programs that were written complying to version 2.0.2 of the Apache Spark library. It might be the case that previous or future libraries of the tool are still compatible, however, any change in the classes mocked up by the surrogate Spark library included in the module might render those programs incompatible. Users are encouraged to modify the surrogate library to match another official Spark release as long as the semantics are preserved.

Analyses using *JPF-SymSpark* are expected to be run on portions of Spark code that contain a series operations applied to a single RDD. Including several RDDs in an execution could provide an invalid outcome. This point is suggested in section 5.3 as one of the possible future extensions of the tool.

Furthermore, there were several concealed, or at least not evident, aspects of SPF that arose during the implementation of *JPF-SymSpark*. These aspects represented major obstacles in the development process and had an impact on the scope of the developed tool. We consider relevant to indicate these pitfalls as part of the evaluation in order to guide future research initiatives on the field and also to improve the knowledge base when assessing SPF in the research context. The intent of these remarks is not to diminish SPF in any sense, on the contrary, it aims to guide future researchers to the weak spots that require more attention.

The most relevant impediment is the limited support of symbolic String operations. Although it has been a work in progress since 2012 [48, 47], there are still some key String operations that are not yet supported by SPF. For example, the *split* operation, which is commonly used in Spark programs, is not supported and if included in an analysis it halts the verification and crashes JPF. Additionally, constraints that combine conditions on the String structure and its length are not solved correctly, bypassing any restrictions set on the size of the String.

Moreover, specialized String constraint solvers are claimed to be supported, however, in practice this is no longer the case. This situation not only applies to String solvers but also to other third-party solvers specialized in more complex numerical constraints. The problem is that the implemented interfaces that communicate with the constraint solvers are outdated given that they were initially implemented based on now obsolete versions of the tools. Some solvers like CVC3 [7] are still compatible (although the newer libraries must be included), while others like Z3 [14] are no longer compatible.

Another relevant obstacle in the context of Spark applications is the limited support of symbolic data structures, sometimes also referred to as symbolic heap or symbolic objects [44]. Although supported, symbolic data structures often generate errors if used in the regular way inside Spark transformations and actions. The lack of a convenient interaction mechanism when dealing with symbolic objects makes it difficult to build extensions on top of it.

Some obstacles were bested by means of a workaround. Such is the case of the lack of support of lambda expressions as target methods to be analyzed by SPF. As noted in a series of posts exchanged between the author and Kasper Luckow (contributor to SPF and author of JDart [35]) in the official JPF forum, the workaround consists in referring to the static methods in the anonymous classes that are generated as a consequence of the compilation of the lambda expression. This solution is further explained in section 1.4.

With a few exceptions, the JPF and SPF communities are relatively silent and the tools seems to be lacking enthusiasts. This plays a big role when trying to extend the current tools; software communities in other open-source projects have a more structured communication mechanism and a clear list of new features and bugs where the collaborators can easily share information.

Lastly, there are two aspects that have to be considered before engaging into an adaptation or extension of SPF: First, SPF has a large codebase of more than a hundred thousand lines building up for more than ten years of development. Tackling such a large codebase takes a considerable amount of time and getting used to the different programming styles makes it more challenging when identifying relevant portions of the code. Second, new versions are seldom released and when they are, they often include a huge number of undocumented changes that are not clearly specified in the revision notes. This makes it particularly hard when tracking differences between the official documentation and the current software.

5 Conclusion

This chapter concludes the work done in this thesis. First, it summarizes the contributions of *JPF-SymSpark* as well as the results of its evaluation. Next, it presents a revisited discussion about the aim and research questions defined at the beginning of this study. Finally, it points towards several improvements to *JPF-SymSpark* and future work on this line of research.

5.1 Summary

This study introduces *JPF-SymSpark*, the first symbolic execution framework for Apache Spark programs. It is built as an extension of Java PathFinder (JPF), a state model checking tool for Java bytecode, and based upon Symbolic PathFinder (SPF), a generic symbolic execution module of JPF. The main goal of the tool is to generate reduced input datasets that offer full path coverage of the program under test. Such datasets can have several uses in the development process of a Spark application, for example, as input data for unit tests.

JPF-SymSpark follows a process that is able to reason over the RDD's API exposed by Spark. This API consists of a set of operations defined as higher-order functions that are applied over collections of data. The tool is capable of correctly interconnecting the symbolic executions of all the relevant functions in a program under test while discarding any unnecessary overhead introduced by the internal Spark logic. This reasoning over the interrelation of Spark operations and the data flow among them is the most useful contribution of this work. To our knowledge, there has not been any study over the application of symbolic execution in big data frameworks.

A twofold evaluation of *JPF-SymSpark* is presented in this work: a qualitative examination of the overall tool and a quantitative appraisal of iterative symbolic executions. The former consists in contrasting the tool against a series of functional requirements of an ideal symbolic execution framework for Apache Spark, while the latter explores the behavior of the iterative reduce strategy in order to highlight performance obstacles when choosing this execution approach. In the sense of the qualitative examination, although *JPF-SymSpark* fulfills the core requirements, the lack of support of symbolic data structures and partial support of symbolic String operations pose as major limitations to the usability of the framework. The quantitative evaluation shows that cumulative symbolic variables translate into more complex path conditions resulting in poor time performance. However, higher numbers of unsatisfiable path conditions allow a faster exploration of the state space of the program under test, given that it is easier for the constraint solver to discard those conditions that are unsatisfiable. The most relevant conclusion of the evaluation is that symbolic constraint solvers pose as the main bottlenecks in terms of performance. Lastly, a discussion on the limitations of the tool and the process under the context of JPF is presented in order to advise similar research interests on the advantages and disadvantages of the chosen supporting platforms.

During the development of the tool, SPF presented unexpected behaviors when analyzing some programs. Most of these abnormal results were caused by common programming practices in Spark applications, for example, the use of anonymous classes and lambda expressions to represent the parameter functions passed to many of Spark's operations. The SPF extension was modified accordingly in order to cope with these scenarios. The modifications done to the SPF module are: the detection of synthetic bridge methods, the consistent ordering of String path conditions and improvements on the visitor pattern applied to symbolic constraints. Some of these modifications were included in a patch and submitted for revision to the SPF administrator. However, to the date this document was published they have not been included in the official source code.

5.2 Aim and Research Questions Revisited

A revision of the aim and research questions defined at the beginning of this work provides an overall understanding of the lessons learned and the results obtained during the execution of this study.

The aim of this work is to determine if *symbolic execution techniques can be used in the context of Apache Spark as a big data framework to generate reduced input data sets that enforce full path coverage*. *JPF-SymSpark* serves as an example of the application of symbolic execution techniques used to generate such datasets in the context of Apache Spark. Nonetheless, the limitations of SPF as the underlying symbolic execution framework scope down the application of the technique to an unsubstantial group of use cases that do not represent the characteristics of most realistic Apache Spark programs. As mentioned earlier, the lack of support of symbolic data structures and symbolic Strings strongly reduces the usability of such a technique. Further improvements in SPF or newer symbolic execution frameworks would improve the applicability of the technique as well as the capacity for more thorough evaluations.

The following answers correspond to the research questions stated as premises of this study:

1. *Is symbolic execution a suitable technique for analyzing programs in the context of Spark applications.*

   Yes. As *JPF-SymSpark* demonstrates, the technique can be used to analyze Apache Spark programs although an additional reasoning on the structure of the operations had to be included in order to produce any valuable result.

2. *What are the particular characteristics associated with the symbolic execution of a Spark program.*

   There are two particular characteristics that differentiate the symbolic execution of Spark programs from other programs. The first is that the control flow instructions of an application can be contained inside functions passed as parameters to Spark operations. Such a situation requires path conditions and symbolic transformations to be percolated between Spark operations. The second characteristic is that some Spark operations, in particular aggregation functions, have control flow semantics defined intrinsically. One example is the *reduce* action, that defines an iterative behavior and accumulates a value resulting from the application of the passed user-defined function. In a symbolic execution, this iterative behavior needs to be considered in order to control undesirable results, as is the case of the symbolic state explosion.

3. *Is there a symbolic execution framework that can be adapted to perform symbolic executions of Spark*

programs.

Spark provides several implementations for different programming languages, being the most common for the Scala and Java programming languages. Both of this languages compile to the Java Virtual Machine. There are only a few symbolic execution frameworks that can be applied to Java programs or their respective compiled bytecode versions. SPF is the most complete framework available and with the most amount of documentation.

4. *If it exists, what are the advantages and disadvantages of such a framework in the context of Spark applications.*

   SPF is a powerful framework that is able to replace the whole instruction set of the Java bytecode for its symbolic counterparts. A wide variety of symbolic operations for all the primitive types are supported as well as a subset of the operations defined for Strings. It takes care of the identification of branching instructions and, with the help of JPF's state generation engine, is capable of exploring the different paths. It also provides a convenient set of mechanisms to consult and manipulate the execution state if needed. This is particularly useful when percolating path conditions among Spark operations.

   However, SPF does not support symbolic data structures and only supports symbolic Strings partially. This presents a major limitation given that most of the realistic Spark programs operate to a certain extent this type of data. Additionally, SPF has undergone several development iterations and its codebase starts to show hints of code decay. Code comments used as a communication mechanisms between developers, frequent code duplication and outdated examples prove to be serious obstacles for the further improvement of the tool.

5.3 Future Work

The next points represent some improvements and extensions to Symbolic PathFinder and *JPF-SymSpark* that would broaden the scope of programs the tools can analyze.

Symbolic execution of data structures

As mentioned in section 4.3, the limited support of symbolic data structures on SPF affected greatly the scope of *JPF-SymSpark*. The use of data structures in Spark programs is a common practice as it is in object-oriented programming languages. Improving SPF to provide better mechanisms to handle and manipulate symbolic data structures would be the first step to eventually analyzing Spark operations defined over data structures.

This will prove particularly useful in the many cases where RDDs are defined over tuples, and the many pair-oriented operations that are characteristic for this kind of structures. One example of a frequently used operation is the *reduceByKey* transformation, that takes pairs matching the same key and applies a reduce function to them.

Support more Spark operations

Giving support to additional Spark operations would improve the usability of the tool. Most of the operations that are not supported yet are implemented over RDDs of *Pair* or other data structures, hence, in order to support these operations it is necessary to support symbolic data structures first. However, once this occurs, the processing logic of many of these operations resembles to many of the already implemented strategies.

Nevertheless, transformations like *join*, *union*, and *intersection* would require new processing strategies given that these transformations work over two potentially symbolic RDDs, something the we do not contemplate in this work.

Provide a Java annotation for analyzing single methods

The initial approach of this work is to provide a blackbox analysis on Spark programs. Any guidance on how the analyses should proceed are communicated via the configuration properties that are defined beforehand for each analysis. However, in some cases it could prove useful to provide the means to indicate in the source code which particular Spark operations should be considered in an analysis.

JPF offers a mechanism to introduce information relevant to an analysis directly in the source code of the program being analyzed through customized Java annotations. A new Java annotation can be created to mark which Spark operations ought to be symbolically executed. Moreover, complementary information about the data being processed could be provided, for example a pre-condition that could be appended to any path condition found during the analysis. Although this will shift the tool to a whitebox approach, it will offer more flexibility for the users.

Support Scala

Scala is a functional programming language that also adopts many object-oriented concepts [52]. The Spark library was originally written in Scala, only supporting other programming languages like Java and Python later on. For this reason, Scala is the default language of choice when writing Spark programs.

By design, Scala is compiled to the Java Virtual Machine in order to take advantage of the portability and the long-time tested stability of Java. Given that Scala works on the same set of bytecode instructions, it would be possible to use JPF (and by extension SPF as well) to analyze Scala. However, there is no official documentation about JPF supporting Scala, although it is often a topic of conversation in the official JPF community. Being able to analyze Spark programs written in Scala will improve greatly the scope of the tool, although it would prove to be a daunting task. It will require first to ensure compatibility with JPF and SPF, to later create the necessary mechanisms that would allow *JPF-SymSpark* to support such programs.

A Appendix - Contributions and Collaborations to SPF

This appendix explains in detail all the modifications done to the Symbolic PathFinder (SPF) extension that were necessary in order to be able to execute Apache Spark programs symbolically. These modifications were submitted as a patch for the SPF project to Corina Păsăreanu, lead developer and repository administrator of SPF. The changes are under revision and could be included in future releases. The modifications are:

- Detection of synthetic bridge methods
- Consistent ordering of String path conditions
- Improvements to the visitor pattern in the symbolic constraints

A.1 Detection of Synthetic Bridge Methods

The *synthetic* modifier is a compiler-only modifier that marks a certain method or class included in the compiled bytecode that was not part of the original source code. Basically it refers to any construct introduced by the compiler. There are several reasons to include a synthetic constructs in the bytecode, for example, dynamic proxy classes or references in switch statements.

Likewise, the *bridge* modifier is a compiler-only modifier that is used to mark a method that simply delegates its invocation to another method, hence serving as a bridge. For example, bridge methods are necessary when implementing generic interfaces. After type erasure, the signatures of the concrete methods implementing an interface defined over a generic type will no longer match with the methods in the interface itself; the methods in the interface will be defined over the *Object* class or the closest class in the hierarchy if the generic type was covariant or countervariant, while the methods in the class implementing the interface will be defined over the concrete type used in the implementation. This problem is solved by introducing bridge methods in the concrete implementation of the interface that match the signatures of the methods in the interface. These methods simply perform a cast in the parameters defined over the generic type and forward the call to the correct method. By definition, bridge methods are always synthetic given that they are constructs introduced by the compiler.

Listing A.1 shows an example where this can be appreciated. This example was chosen because of its resemblance to the way Spark operations are implemented. On line 2 an internal interface is defined over a generic type *T*. On line 5 the *sampleMethod* is defined which takes an implementation of the aforementioned interface over the concrete *Interger* type. This method invokes the *call* method of the parameter interface with a concrete value. In lines 9 to 13 the interface is implemented over the *Integer* type as an anonymous class and it is passed directly to an invocation of the *sampleMethod* method. The code presented in the example is correct and it compiles without problems. However, it requires the inclusion of a *synthetic bridge* methods that fills the gap for those methods that do not match the signatures

```java
public class SymbolicGenericTest {
  interface SampleInterface<T> {
    public T call(T param);
  }
  public static Integer sampleMethod(SampleInterface<Integer> a) {
    return a.call(2);
  }
  public static void main(String[] args){
    sampleMethod(new SampleInterface<Integer>() {
      @Override
      public Integer call(Integer param) {
        return param+1;
      }
    });
  }
}
```

Listing A.1: Example of a program where a symbolic bridge method will be added after commpilation. The *SampleInterface* is programmed as an interface over a generic type. Anytime this interface is implemented with a concrete type, an intermediate bridge method is created to cast the invocation of the method in the most generic type to the type specified in the concrete implemention.

after type erasure. Listing A.2 shows how the program in listing A.1 would look after type erasure.

Although the program shown in listing A.2 is not a valid Java program, it serves to illustrate the purpose of a *synthetic bridge* method after type erasure occurs during the compilation phase. Line 2 shows the definition of the internal interface but this time the generic type has been replaced by the closest class higher in the hierarchy; the *Object* class. Moreover, the method *sampleMethod* defined in line 5 does not assume that the passed object is an implementation of the interface over the *Integer* type and, because of this, it requires both its return value and the parameters to be casted to the corresponding types matching the signature of the interface and the return type of the method itself. Lastly, lines 9 to 16 contain the implementation of the interface, however, the overridden method works now as a bridge method between the concrete implementation of type *Integer*.

This kind of code is common among Spark programs to specify the functions passed to the actions and transformations. Most of Spark operations depend on functions that are defined based on several functional generic interfaces, for example, the *Function* interface used in the *filter* transformation and the *Function2* interface used in the *reduce* action. Disregarding whether these functions are implemented as anonymous classes or lambda expressions, the existence of symbolic bridge methods will always be present as a consequence of type erasure.

```java
public class SymbolicGenericTest {
  interface SampleInterface {
    public Object call(Object param);
  }
  public static Integer sampleMethod(SampleInterface a) {
    return (Integer)a.call((Object)2);
  }
  public static void main(String[] args){
    sampleMethod(new SampleInterface() {
      @Override
      public synthetic bridge Object call(Object param) {
        return (Object)call((Integer)param);
      }
      public Integer call(Integer param) {
        return param+1;
      }
    });
  }
}
```

Listing A.2: This is the same sample program shown in listing A.1 but showing how it would look after type erasure. This sample, although not a compiling Java program, illustrates the necessity of *synthetic bridge* methods to ensure correct inheritance after type erasure.

Let us assume that we would like SPF to carry out a symbolic execution of the method *call*. Then, the *symbolic.method* property in the *.jpf* file should point to:

```
symbolic.method= SymbolicGenericTest$1.call(sym)
```

The relevant method invocation in the analysis would be the one defined over the *Integer* type which is the one that actually has an implementation. However, the *symbolic.method* property does not provide any information about the types of the symbolic parameters; it might try to analyze a method matching the method name and class but defined over the *Object* class, as well as one defined over the *Integer* type.

This occurs in line 6 of listing A.1 when the method *call* is invoked. This line triggers an `INVOKEINTERFACE` instruction that attempts to invoke the *synthetic bridge call* method. At this point, SPF detects a method invocation that matches the specified property and attempts to set the symbolic variables. Nevertheless, under this circumstances SPF always stopped the analysis and aborted the execution due to an unhandled exception relative to empty method arguments.

Considering that *synthetic bridge* methods should not be relevant to any symbolic execution and having detected that the faulty behavior only occurred when *synthetic bridge* methods were found, we resolved that only methods who did not contain the *synthetic* nor *bridge* modifiers should be considered as valid targets for the analysis.

For this purpose a new validation was included in the *BytecodeUtils* class of SPF. This validation consists of comparing the modifiers of the invoked method an determining if they contain the *0x0040* and *0x1000* values as specified in the official specification of the JVM [33]. After implementing the change, the symbolic execution of methods in this scenario worked as expected.

A.2 Order of String Path Conditions

This change is rather simple. It aims to maintain consistency on the way regular path conditions and String path condition explore their options. Although we consider that both path conditions should be refactored in order to respond to a common API, a temporary solution to this problem was more efficient in terms of time and resources.

The basic idea behind of this modification was to switch the ways String path conditions were explored; the path that evaluated to *false* first followed by the path that evaluated to *true*. This was helpful when dealing with some strategies, for example in the case of the filter strategy (see figure 3.5) where the negative branch triggers an immediate break in the state transition. If this change had not been included, a verbose check would have been necessary on every instance were a path condition would have been processed.

A.3 Improving the Visitor Pattern in the Symbolic Constraints

SPF implements both symbolic constraints and symbolic expressions following the visitor pattern [17]. Symbolic expressions are used to represent any transformation of a symbolic value during the execution. They contain a left operand, a right operand, and an operator; the operands have to be symbolic expressions as well. When a visitor is used to explore the data structure, first the current element is visited and then the left and right operands are visited respectively.

On the other hand, symbolic constraints are used to represent boolean expressions evaluated on symbolic values. Constraints are used to produce path conditions, which in turn are later parsed and passed to the solvers to determine their satisfiability. On the contrary to what it could be expected, symbolic constraints were not implemented in the same way as the symbolic expressions. They contain also a left and right operand, and an operator but, additionally they maintain a reference to a chained constraint kept in an attribute called *and*.

The implementation of the visitor pattern in the case of the constraints was forwarded to the left and right operands but not to the following constraints referenced by the *and* attribute. This resulted in incomplete visits, not being able to explore completely all the constraints in a path condition. This was particularly necessary in the case of the iterative reduce strategy (see figure3.7b), when the path condition prior to the *reduce* method was cloned to match the new symbolic variable generated each iteration.

The modification in this case was trivial. If the *and* attribute of the constraint was not null then it would accept a visitor prior to the left and right operands. This behaves similarly to a depth-first search.

B Appendix - Installation and Use

This appendix aims to serve as a checklist for installing *JPF-SymSpark*, as well as a brief guide on how to use the module in combination with JPF. There are two installation approaches:

- Docker approach (preferred)
- Manual approach

B.1 Docker approach

Docker is a widely supported platform for the creation and maintenance of virtual containers [40]. It is designed to provide self-contained, portable environments that are ideal for distributing software with a fixed set of dependencies.

For this reason, we provide the description of a Docker container that prepares an environment with all the dependencies and configurations required by *JPF-SymSpark*. This installation method is the preferred approach given its simplicity and tested behavior. The following instructions guide the process for creating the Docker container; Docker is assumed to be installed already.

1. Clone the *JPF-SymSpark* repository

   ```
   git clone https://github.com/omrsin/jpf-symbc.git
   ```

2. Go to the root directory of the project and build the container

   ```
   docker build -t jpf-symspark .
   ```

3. Once the container has been successfully built, run a container shell

   ```
   docker run -it jpf-symspark
   ```

4. Inside the container, the installation of the module can be validated by running

   ```
   cd jpf-symspark/src/examples/de/tudarmstadt/thesis/symspark/examples/java/applied/
   jpf WordCountExample.jpf
   ```

The output should display the outcome of the analysis run on the WordCountExample.java program.

The created container could serve as a template for any future projects that aim to execute analyses on Spark programs.

B.2 Manual approach

This section lists all the dependencies and configuration requirements that are needed to execute JPF and *JPF-SymSpark* correctly. By no means it should be considered as an extensive guide that works under all platforms; the described approach will only be focused on explaining the steps used in the environment where to project was developed. The following steps should be executed in an Ubuntu 16.04 Desktop OS [53] with at least 2 gigabytes of memory.

Prerequisites

The following dependencies need to be installed first:

- **Java**: Preferably the oracle distribution. The version used was *1.8.0_111*.

- **Mercurial**: Used by JPF as a version control tool [39]. Can be installed with the regular package manager. The version used was *3.7.3*.

- **Apache Ant**: Build tool for Java [3]. Can be installed with the regular package manager. The version used was *1.9.6*.

- **JUnit**: Framework for unit testing in Java [28]. Can be downloaded from the official website and placed in a known directory. The version used was *4.12*.

jpf-core

This is the main module of JPF. In addition to the following installation steps, always check the official installation instructions because the repositories could have been moved.

1. Clone the project *jpf-core* from the official repository

   ```
   hg clone http://babelfish.arc.nasa.gov/hg/jpf/jpf-core
   ```

2. Create the *site.properties* file as suggested in the official JPF site. Make sure to be pointing the jpf-core property to the directory where the project was cloned. Additionally, be sure to remove or comment the other references to modules in the example *.properties* file provided.

3. In order to be able to build the project, JUnit libraries need to be in the classpath. The ant script requires the JUNIT_HOME directory to be specified. This could be done by creating the JUNIT_HOME environment variable and placing it in the path. However, the *build.xml* file of the *jpf-core* project could be modified by replacing the value property *junit.home* with the value of the directory where the JUnit library was placed. This option is more convenient because it avoids polluting the classpath with variables that might potentially generate conflicts with other programs.

4. Finally, build the *jpf-core* project. Go to the directory where it is located and execute

   ```
   ant test
   ```

5. In order to test if the build was successful, go to the jpf-core directory and execute

   ```
   java -jar build/RunJPF.jar src/examples/Racer.jpf
   ```

   JPF should run a basic model checking analysis on the Racer example.

jpf-symbc

This is the symbolic execution module built on top of JPF known as SPF or Symbolic Pathfinder. In addition to the following installation steps, always check the official installation instructions. The version used in this project is a modified version of the module. At the moment of this publication, the repository of this customized version might not be public yet.

1. Clone the project *jpf-symbc* from our hosted repository on the same root directory where *jpf-core* was cloned

   ```
   git clone https://github.com/omrsin/jpf-symbc.git
   ```

2. Update the *site.properties* file as suggested in the official JPF site. Every time a new module is downloaded this file must be updated. Make sure to be pointing the *jpf-symbc* property to the directory where the project was cloned.

3. Set the JUNIT_HOME environment variable or update the *build.xml* file of the project in a similar fashion as explained for *jpf-core*.

4. Build the *jpf-symbc* project. Go to the directory where it is located and execute

   ```
   ant test
   ```

   If the test target generates errors then executing `ant build` would be sufficient (considering no build errors were found). With this, SPF should be available.

jpf-symspark

This is the module that we developed based on SPF. It provides the mechanisms to carry out symbolic executions of Spark programs. The installation steps follow the same pattern as in the case of *jpf-symbc*.

1. Clone the *jpf-symspark* project from our hosted repository on the same root directory where *jpf-core* and *jpf-symbc* were cloned

   ```
   https://github.com/omrsin/jpf-symspark.git
   ```

2. Update the *site.properties* file as suggested in the official JPF site. Every time a new module is downloaded this file must be updated. Make sure to be pointing the *jpf-symspark* property to the directory where the project was cloned.

3. Build the *jpf-symspark* project. Go to the directory where it is located and execute

   ```
   ant build
   ```

4. In order to test if the build was successful, go to the jpf-symspark directory and execute

   ```
   java -jar build/RunJPF.jar \\
   src/examples/de/tudarmstadt/thesis/symspark/examples/java/applied/WordCountExample.jpf
   ```

   This will execute an analysis on an example program and produce a reduced input dataset that explores all possible paths.

It is recommended to install the *eclipse-jpf* plug-in for the Eclipse IDE. This tool enables the IDE to carry out analyses as specified in the *.jpf* files. To install it simply follow the instructions as described in the official JPF website.

B.3 Usage

The *JPF-SymSpark* module is used in a similar way as SPF. However, some additional properties were added to the *.properties* file of the module or must be included in the *.jpf* file used for a particular analysis in order execute correctly.

The properties used are:

- **spark.methods**: Used to indicate which spark operations are to be analyzed by the module. Every time a Spark operation with this name is found in the program it will be executed symbolically. This option replaces the use of the *symbolic.method* property used by SPF given that the target methods to be analyzed will be set dynamically during the analysis based on the spark operations defined. The values supported for this property are: *filter*, *map*, *reduce* and *flatMap*; multiple values must be separated with a semicolon.

- **spark.reduce.iterations**: Used to indicate how many iterations of a reduce action will be analyzed. The value must be greater than 0. This option is only relevant if the reduce action was included among the methods to be analyzed in the *spark.methods* property.

- **listener**: Defines the listener to be used during the analysis. By default, the module uses the *Spark-MethodListener* as defined in its *.properties* file.

- **jvm.insn_factory.class**: This property is used to specify the instruction factory used by the module.

```
1  @using=jpf-symspark
2
3  jpf-spark.package_path=
       de.tudarmstadt.thesis.symspark.examples.java.applied
4
5  target=${jpf-spark.package_path}.WordCountExample
6  target.args=input,output
7
8  symbolic.dp=choco
9  symbolic.string_dp=automata
10
11 spark.methods=filter
```

Listing B.1: Sample *.jpf* file of the *JPF-SymSpark* module corresponding to the WordCountExmaple.java program.

By default, the module uses the *SparkSymbolicInstructionFactory* as defined in the *.properties* file.

Other SPF specific properties are still supported. Listing B.1 shows the *.jpf* file used in the *WordCountExample* analysis. The property *jpf-spark.package_path* defined in line 3 is just a shorthand for the full path where the target file is found; it is not mandatory for any analysis as long as the target is defined correctly. Additionally, line 11 defines which methods are to be analyzed; in this case just the *filter* transformation is relevant for the analysis.

The analysis can be run by means of the previously mentioned Eclipse plug-in or by execution through the command line as shown in the installation steps. Additional examples can be found in the *examples* directory of the *JPF-SymSpark* module.

List of Figures

List of Tables

List of Listings

Bibliography

[1] Allen, F. E. “Control Flow Analysis”. In: *ACM SIGPLAN Notices* 5.7 (July 1970), pp. 1–19. ISSN: 0362-1340. DOI: 10.1145/390013.808479.

[2] Anand, S., Păsăreanu, C. S., and Visser, W. “JPF–SE: A Symbolic Execution Extension to Java PathFinder”. In: *Tools and Algorithms for the Construction and Analysis of Systems: 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007. Proceedings*. Ed. by Grumberg, O. and Huth, M. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 134–138. ISBN: 978-3-540-71209-1. DOI: 10.1007/978-3-540-71209-1_12.

[3] *Apache Ant*. URL: http://ant.apache.org/ (visited on 2017).

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

[5] Armbrust, M. et al. “Scaling Spark in the Real World: Performance and Usability”. In: *Proceedings of the VLDB Endowment* 8.12 (Aug. 2015), pp. 1840–1843. ISSN: 2150-8097. DOI: 10.14778/2824032.2824080.

[6] Armbrust, M. et al. “Spark SQL: Relational Data Processing in Spark”. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’15. Melbourne, Victoria, Australia: ACM, 2015, pp. 1383–1394. ISBN: 978-1-4503-2758-9. DOI: 10.1145/2723372.2742797.

[7] 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.

[8] Bush, W. R., Pincus, J. D., and Sielaff, D. J. “A static analyzer for finding dynamic programming errors”. In: *Software: Practice and Experience* 30.7 (2000), pp. 775–802. ISSN: 1097-024X. DOI: 10.1002/(SICI)1097-024X(200006)30:7<775::AID-SPE309>3.0.CO;2-H.

[9] Cadar, C., Dunbar, D., and Engler, D. “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*. OSDI’08. San Diego, California: USENIX Association, 2008, pp. 209–224.

[10] Cadar, C. and Sen, K. “Symbolic Execution for Software Testing: Three Decades Later”. In: *Communications of the ACM* 56.2 (Feb. 2013), pp. 82–90. ISSN: 0001-0782. DOI: 10.1145/2408776.2408795.

[11] 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.

[12] Csallner, C., Fegaras, L., and Li, C. "New Ideas Track: Testing Mapreduce-style Programs". In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. ESEC/FSE '11. Szeged, Hungary: ACM, 2011, pp. 504–507. ISBN: 978-1-4503-0443-6. DOI: 10.1145/2025113.2025204.

[13] Csallner, C., Tillmann, N., and Smaragdakis, Y. "DySy: Dynamic Symbolic Execution for Invariant Inference". In: *Proceedings of the 30th International Conference on Software Engineering*. ICSE '08. Leipzig, Germany: ACM, 2008, pp. 281–290. ISBN: 978-1-60558-079-1. DOI: 10.1145/1368088.1368127.

[14] De Moura, L. and Bjørner, N. "Z3: An Efficient SMT Solver". In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS'08/ETAPS'08. Budapest, Hungary: Springer-Verlag, 2008, pp. 337–340. ISBN: 3-540-78799-2, 978-3-540-78799-0.

[15] Dean, J. and Ghemawat, S. "MapReduce: Simplified Data Processing on Large Clusters". In: *Communications of the ACM* 51.1 (Jan. 2008), pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492.

[16] *Engine health management*. Rolls-Royce. URL: http://www.rolls-royce.com/about/our-technology/enabling-technologies/engine-health-management.aspx#analyse (visited on 2017).

[17] Gamma, E. et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1st ed. Addison-Wesley Professional, 1994. ISBN: 0201633612.

[18] Ghemawat, S., Gobioff, H., and Leung, S.-T. "The Google File System". In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. SOSP '03. Bolton Landing, NY, USA: ACM, 2003, pp. 29–43. ISBN: 1-58113-757-5. DOI: 10.1145/945445.945450.

[19] 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*. PLDI '05. Chicago, IL, USA: ACM, 2005, pp. 213–223. ISBN: 1-59593-056-6. DOI: 10.1145/1065010.1065036.

[20] Godefroid, P., Levin, M. Y., and Molnar, D. A. "Automated Whitebox Fuzz Testing". In: *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*. 2008.

[21] *Google Zeitgeist 2012: A Year in Search*. Google. URL: http://www.google.com/zeitgeist/2012/#the-world.

[22] Gosling, J. et al. *The Java Language Specification, Java SE 8 Edition*. 1st. Addison-Wesley Professional, 2014. ISBN: 013390069X, 9780133900699.

[23] Gulzar, M. A. et al. "BigDebug: Debugging Primitives for Interactive Big Data Processing in Spark". In: *Proceedings of the 38th International Conference on Software Engineering*. ICSE '16. Austin, Texas: ACM, 2016, pp. 784–795. ISBN: 978-1-4503-3900-1. DOI: 10.1145/2884781.2884813.

[24] Havelund, K. and Pressburger, T. "Model checking JAVA programs using JAVA PathFinder". In: *International Journal on Software Tools for Technology Transfer* 2.4 (2000), pp. 366–381. ISSN: 1433-2779. DOI: 10.1007/s100090050043.

[25] Hoare, C. A. R. "An Axiomatic Basis for Computer Programming". In: *Communications of the ACM* 12.10 (Oct. 1969), pp. 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259.

[26] Interlandi, M. et al. "Titian: Data Provenance Support in Spark". In: *Proceedings of the VLDB Endowment* 9.3 (Nov. 2015), pp. 216–227. ISSN: 2150-8097. DOI: 10.14778/2850583.2850595.

[27] Isard, M. et al. "Dryad: Distributed Data-parallel Programs from Sequential Building Blocks". In: *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. EuroSys '07. Lisbon, Portugal: ACM, 2007, pp. 59–72. ISBN: 978-1-59593-636-3. DOI: 10.1145/1272996.1273005.

[28] *JUnit*. URL: http://junit.org (visited on 2017).

[29] *JaCoCo Java Code Coverage Library*. Mountainminds GmbH & Co. URL: http://www.eclemma.org/jacoco/ (visited on 2017).

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

[31] Khurshid, S., Păsăreanu, C. S., and Visser, W. "Generalized Symbolic Execution for Model Checking and Testing". In: *Tools and Algorithms for the Construction and Analysis of Systems: 9th International Conference, TACAS 2003 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003 Warsaw, Poland, April 7–11, 2003 Proceedings*. Ed. by Garavel, H. and Hatcliff, J. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 553–568. ISBN: 978-3-540-36577-8. DOI: 10.1007/3-540-36577-X_40.

[32] King, J. C. "Symbolic Execution and Program Testing". In: *Communincations of the ACM* 19.7 (July 1976), pp. 385–394. ISSN: 0001-0782. DOI: 10.1145/360248.360252.

[33] Lindholm, T. et al. *The Java Virtual Machine Specification, Java SE 8 Edition*. 1st. Addison-Wesley Professional, 2014. ISBN: 013390590X, 9780133905908.

[34] Luckow, K. S. and Păsăreanu, C. S. "Symbolic PathFinder V7". In: *SIGSOFT Software Engineering Notes* 39.1 (Feb. 2014), pp. 1–5. ISSN: 0163-5948. DOI: 10.1145/2557833.2560571.

[35] Luckow, K. et al. "JDart: A Dynamic Symbolic Analysis Framework". In: *Tools and Algorithms for the Construction and Analysis of Systems: 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. Ed. by Chechik, M. and Raskin, J.-F. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 442–459. ISBN: 978-3-662-49674-9. DOI: 10.1007/978-3-662-49674-9_26.

[36] Marr, B. *How Big Data Drives Success At Rolls-Royce*. 2015. URL: https://www.forbes.com/sites/bernardmarr/2015/06/01/how-big-data-drives-success-at-rolls-royce/#1ce668951d69 (visited on 2017).

[37] McCabe, T. J. "A Complexity Measure". In: *Proceedings of the 2Nd International Conference on Software Engineering*. ICSE '76. San Francisco, California, USA: IEEE Computer Society Press, 1976, pp. 407–.

[38] Meng, X. et al. "MLlib: Machine Learning in Apache Spark". In: *Journal of Machine Learning Research* 17.1 (Jan. 2016), pp. 1235–1241. ISSN: 1532-4435.

[39] *Mercurial, Source Control Management*. URL: https://www.mercurial-scm.org/ (visited on 2017).

[40] Merkel, D. "Docker: Lightweight Linux Containers for Consistent Development and Deployment". In: *Linux Journal* 2014.239 (Mar. 2014). ISSN: 1075-3583.

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

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

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

[44] Păsăreanu, C. S. and Rungta, N. "Symbolic PathFinder: Symbolic Execution of Java Bytecode". In: *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. ASE '10. Antwerp, Belgium: ACM, 2010, pp. 179–180. ISBN: 978-1-4503-0116-9. DOI: 10.1145/1858996.1859035.

[45] Păsăreanu, C. S. and Visser, W. "Symbolic Execution and Model Checking for Testing". In: *Hardware and Software: Verification and Testing: Third International Haifa Verification Conference, HVC 2007, Haifa, Israel, October 23-25, 2007. Proceedings*. Ed. by Yorav, K. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 17–18. ISBN: 978-3-540-77966-7. DOI: 10.1007/978-3-540-77966-7_5.

[46] 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. Seattle, WA, USA: ACM, 2008, pp. 15–26. ISBN: 978-1-60558-050-0. DOI: 10.1145/1390630.1390635.

[47] Păsăreanu, C. S. et al. "Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis". In: *Automated Software Engineering* 20.3 (2013), pp. 391–425. ISSN: 1573-7535. DOI: 10.1007/s10515-013-0122-2.

[48] Redelinghuys, G. "Symbolic String Execution". Master Thesis. University of Stellenbosch, 2012.

[49] 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*. ISSTA '06. Portland, Maine, USA: ACM, 2006, pp. 157–168. ISBN: 1-59593-263-1. DOI: 10.1145/1146238.1146256.

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

[51] Souza, M. et al. "CORAL: Solving Complex Constraints for Symbolic PathFinder". In: *NASA Formal Methods: Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*. Ed. by Bobaru, M. et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 359–374. ISBN: 978-3-642-20398-5. DOI: 10.1007/978-3-642-20398-5_26.

[52] *The Scala Programming Language*. URL: https://www.scala-lang.org/ (visited on 2017).

[53] *The leading operating system for PCs, IoT devices, servers and the cloud | Ubuntu*. Canonical Ltd. URL: https://www.ubuntu.com/desktop (visited on 2017).

[54] 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. London, United Kingdom: ACM, 2007, pp. 97–107. ISBN: 978-1-59593-734-6. DOI: 10.1145/1273463.1273478.

[55] Venkataraman, S. et al. "SparkR: Scaling R Programs with Spark". In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD '16. San Francisco, California, USA: ACM, 2016, pp. 1099–1104. ISBN: 978-1-4503-3531-7. DOI: 10.1145/2882903.2903740.

[56] Visser, W., Păsăreanu, C. S., and Khurshid, S. "Test Input Generation with Java PathFinder". In: *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA '04. Boston, Massachusetts, USA: ACM, 2004, pp. 97–107. ISBN: 1-58113-820-2. DOI: 10.1145/1007512.1007526.

[57] Visser, W. et al. "Model Checking Programs". In: *Automated Software Engineering* 10.2 (2003), pp. 203–232. ISSN: 1573-7535. DOI: 10.1023/A:1022920129859.

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

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

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

[61] 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. New York, New York: ACM, 2013, 2:1–2:6. ISBN: 978-1-4503-2188-4. DOI: 10.1145/2484425.2484427.

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

[63] 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.

[64] Zaharia, M. et al. "Discretized Streams: Fault-tolerant Streaming Computation at Scale". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: ACM, 2013, pp. 423–438. ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522737.