Symbolic Execution of Apache Spark Programs

Omar A. Erminy Ugueto

April 7, 2017

TECHNISCHE
UNIVERSITÄT
DARMSTADT

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

Master Thesis
Topic: Symbolic Execution of Apache Spark Programs

Submitted: April 7, 2017

Supervisor: Prof. Dr. Guido Salvaneschi

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

Abstract

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

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

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

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

Contents

1 Introduction

2 Related Work

2.1 Apache Spark

Spark is a distributed data processing framework that was first introduced in 2012 [38]. Similar to other systems, such as MapReduce [11] and Dryad [16], 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 [30]. 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 Hadoop Distributed File System (HDFS) [34]. Spark converts each line in the file to a *String* element in the newly created RDD. In lines 2 to 4, a chain of transformations is applied to the RDD: First, elements not containing the text "ERROR" are filtered. Next, the remaining elements are transformed to tuples consisting of a certain property (e.g., a time stamp; assumed to be the first information in a log entry) and the number 1. Finally, the tuples are grouped and counted based on the chosen property. Line 5 represents the action applied to the RDD, in this case, saving it to persistent storage.
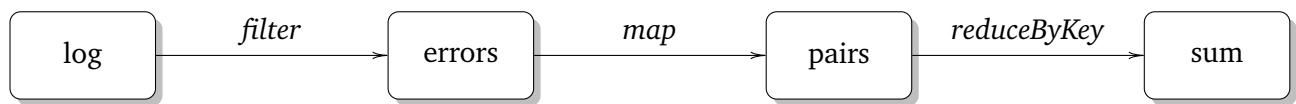
```scala
1 val log = spark.textFile("*file*")
2 val errors = log.filter(_.contains("ERROR"))
3   .map(error => (error.split('\t')(0),1))
4   .reduceByKey(_+_)
5 errors.save()
```
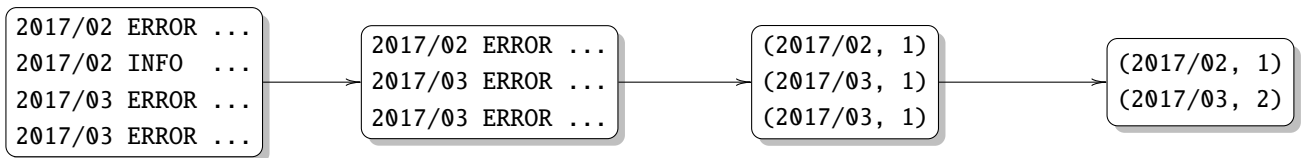
Listing 2.1: Entries in a log file are filtered, grouped and counted based on a common property. Finally the result is saved to persistent storage.

(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.

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 on the reuse of datasets and avoids having to perform costly I/O operations for every iteration. The latter, allows data mining techniques to synthesize queries faster by keeping working data at hand.

Spark is part of the Apache Software Foundation and it is offered as an open-source software [35, 3]. Several purpose-specific libraries are built on top of Spark, as is the case of: MLlib for machine learning [22],

GraphX for graph computations [36], Spark Streaming for stream processing [39], and Spark SQL, an SQL-like interface for structured querying in Spark [4].

In 2014, Spark reported the fastest Daytona GraySort as defined by the Sort Benchmark committee, and later in 2016, Spark was part of the technology stack that claimed the most resource-efficient Daytona CloudSort as defined by the same committee [28, 37, 33]. 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 [24].

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 is read.

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

```
1  public static int search(int[] a, int elem) {
2    for(int i = 0; i < a.length; i++) {
3      if(a[i] == elem) {
4        return i;
5      }
6    }
7    return -1;
8  }
```

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 a -1 is returned.

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

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

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

2.2.2 Symbolic Execution

The first discussions of symbolic execution date several decades back [15, 19]. 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,
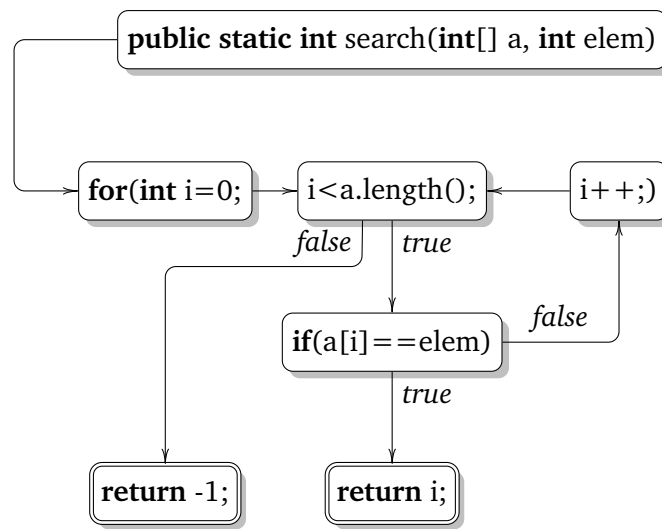
Figure 2.2: Control Flow Graph corresponding to the linear search algorithm shown in listing 2.2. The entry node is the signature of the method, while end nodes, represented with a double frame, contain the return statements that put at end to the execution. The *for* loop instruction was split into its composing statements to better display how the control flow work for this instruction.

in contrast to what would be a concrete value. Throughout the execution, the symbolic values are operated, which generates more complex symbolic expressions. Moreover, control flow statements define logical predicates that could depend on symbolic expressions, bridging the representation of the program from its operational view to a series of logical expressions.

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

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

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

Listing 2.3: Trivial program to illustrate how symbolic execution works.

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 [7].

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 [8], test case generation [6, 9, 12, 31] and static detection of errors [5, 29], among many others [10, 27].
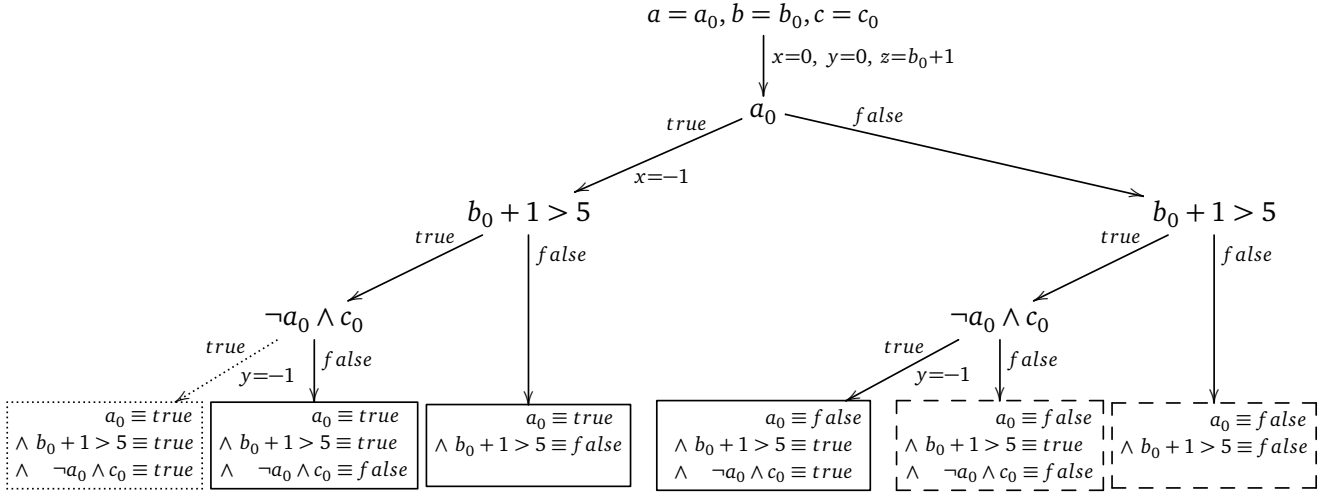
2.3 Java PathFinder

Developed at JPF acr:nasa's Ames Research Center [23], Java PathFinder (JPF) is an execution environment for verification and analysis of Java bytecode programs [32, 17]. Since its publication in the year 2000 [14], 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 [13]. Programs written in Java are compiled to the standardized instruction set of the Java Virtual Machine (JVM), known as Java bytecode. This process makes Java programs portable between architectures implementing the JPF acr:jvm specification. A JPF acr:jvm implementation serves as an interpreter of Java bytecode and allows the optimization and execution of the program tailored for the host platform [20].

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

In its core, JPF is a Java Virtual Machine implemented in Java itself, comprised of several extensible components that dictate the verification strategy to be followed. The fact that JPF is written in Java means

$$a = a_0, b = b_0, c = c_0$$

$$\big\downarrow x{=}0,\ y{=}0,\ z{=}b_0{+}1$$

$$a_0$$

$true$ $x{=}{-}1$ $false$

$$b_0 + 1 > 5 \qquad\qquad\qquad\qquad\qquad b_0 + 1 > 5$$

$true$ $false$ $true$ $false$

$$\neg a_0 \wedge c_0 \qquad\qquad\qquad\qquad\qquad\qquad \neg a_0 \wedge c_0$$

$true$ $false$ $true$ $false$
$y{=}{-}1$ $y{=}{-}1$

| $a_0 \equiv true$ | $a_0 \equiv true$ | $a_0 \equiv true$ | $a_0 \equiv false$ | $a_0 \equiv false$ | $a_0 \equiv false$ |
| $\wedge\ b_0 + 1 > 5 \equiv true$ | $\wedge\ b_0 + 1 > 5 \equiv true$ | $\wedge\ b_0 + 1 > 5 \equiv false$ | $\wedge\ b_0 + 1 > 5 \equiv true$ | $\wedge\ b_0 + 1 > 5 \equiv true$ | $\wedge\ b_0 + 1 > 5 \equiv false$ |
| $\wedge\quad \neg a_0 \wedge c_0 \equiv true$ | $\wedge\quad \neg a_0 \wedge c_0 \equiv false$ | | $\wedge\quad \neg a_0 \wedge c_0 \equiv true$ | $\wedge\quad \neg a_0 \wedge c_0 \equiv false$ | |

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 statements executed following one of the evaluations. More importantly, the leaves collect the predicates that form the path condition for that particular execution. The left most leave in the dotted frame contains an unfeasible path condition given that the predicate cannot be satisfied. Moreover, the path conditions in the dashed frames define executions that will fail the assertion of line 7.

that it is executed on a canonical JPF acr:jvm; in other words, a JPF acr:jvm on top of a JPF acr:jvm.

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

Figure 2.4 portrays the elements that participate in a verification process using JPF . The program under test is loaded into JPF 's core, where its instructions are executed one by one until an execution choice is found. At this point, JPF records the current state and attempts to resume execution, exploring all possible scenarios based on the choice criteria. Once a chosen path has been completely explored, JPF backtracks to a recorded state, in order to explore a new path.

Listing 2.4 introduces an example that illustrates better how JPF acr:jpf works. The program analyzed represents a trivial division of two random values. However, the problem relies on the fact that, under some specific values, the operation could yield invalid. Problems like this, where computations depend
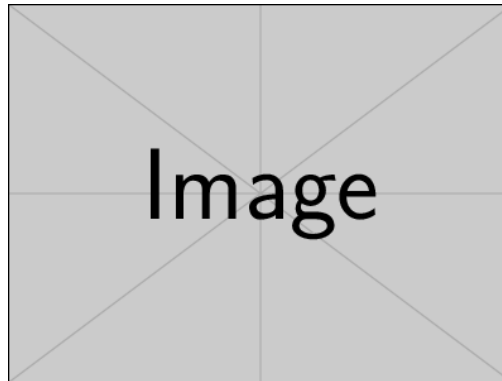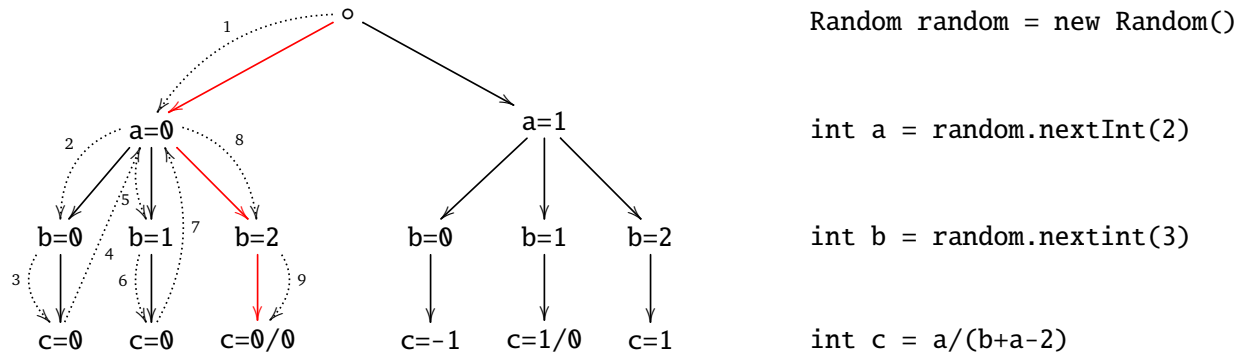
Figure 2.4: JPF Workflow

```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 [23])

on random and unbounded values, are common sources of bugs in real software and, in many cases, are difficult to identify. With the right configuration, JPF acr:jpf could detect this kind of problems by exploring the range of possible values that a random integer could take. Lines 6 and 7 indicate that random values have been generated; at this point JPF acr:jpf could start exploring all different possible 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 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. With a modular design, users of the tool are capable of tuning JPF up to the needs of a wide variety of analyses and verifications. The main extension points are:

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

```
                         o                      Random random = new Random()
              1


        a=0                    a=1              int a = random.nextInt(2)
    2         8


  b=0   b=1   b=2       b=0   b=1   b=2          int b = random.nextint(3)
    5   7
  3   4   6       9


  c=0   c=0   c=0/0     c=-1  c=1/0  c=1         int c = a/(b+a-2)
```

Figure 2.5: State space exploration of the program shown in listing 2.4. JPF starts checking the state space whenever the the conditions that trigger the property to be validated are found; in this case, using random values. The `nextInt` instruction causes JPF to register a *Choice Generator* and start exploring the state space of the possible options. The dashed edges represent the search strategy used to explore the state space; in this case depth-first search. If a given execution path gets to an end and no unexpected behavior is found, JPF backtracks to the latest instruction where a *Choice Generator* was registered and tries a different value. The red arrows point to an execution that triggers an error. Whenever an error is found JPF halts the validation and reports its findings.

operations on symbolic values).

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

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

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

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

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

most likely to detect an anomaly.

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

2.3.1 Symbolic PathFinder

(Mention previous versions of JPF) [18, 26, 2]

(Definition) SPF is part of the Java PathFinder verification tool-set [4]. Java Pathfinder includes JPF-core, an explicit-state model checker, and several extension projects, one of them being SPF (jpf-symbc Java project). The model checker consists of an extensible custom Java Virtual Machine (VM), state storage and backtracking capabilities, different search strate- gies, as well as listeners for monitoring and influencing the search. JPF-core executes the program concretely based on the standard semantics of the Java. In contrast, SPF replaces the concrete execution semantics of JPF-core with a non-standard symbolic interpreta- tion. SPF relies on the JPF-core framework to systemati- cally explore the different symbolic execution paths, as well as different thread interleavings. To limit the possibly in- finite search space that results from symbolically executing programs with loops or recursion, a user-specified depth is provided. We describe SPF's features below. [25]

(Types supported) SPF handles inputs and opera- tions on booleans, integers, reals, and complex data structures, as well as multi-threading, via integration with the Java PathFinder (JPF) model checker [21]

(Symbolic Execution) SPF replaces the standard concrete execution semantics by using a Symbol- icInstructionFactory, that extends the bytecode instruc- tions to manipulate symbolic values and expressions. For ex- ample, when adding two symbolic integers sym1 and sym2 (by executing the IADD bytecode) the result is a symbolic expression representing sym1 +sym2. Storage of symbolic values and expressions is accomplished by assigning symbolic attributes to variables, fields, and stack operands. [25]

(Conditionals and Choice Generators) The symbolic execution of conditional instructions (if statements) involves exploration of two paths corresponding to the branch predicate eval- uating to true and false; both choices are generated non- deterministically by the PCChoiceGenerator. Each generated choice is associated with a path condition encoding the condition and its negation respectively. [25]

(Solvers) The path conditions are checked for satisfiability using off-the-shelf decision pro- cedures or constraint solvers. If the path condition is satisfi- able, the search continues; otherwise, the search back-tracks (meaning that branch is unreachable). SPF uses multiple decision procedures and constraint solvers through a generic interface. Currently, SPF supports: choco for in- teger/real constraints, cvc3 for linear constraints, and the interval arithmetic solver IASolver. Adding support for ad- ditional constraint solvers such as HAMPI and YICES is work in progress. [25]

(Solvers) solvers such as CHOCO [3], CORAL [8], and CVC3 [1] [21]

(Input Data Structures - maybe not needed) SPF uses lazy initialization [5] to handle unbounded input data structures. The execution starts on data structures with un-initialized fields and it initializes them lazily, when the fields are first accesed. A field of class T is initialized non-deterministically to (1) null, (2) a reference to a new instance of class T with uninitialized fields, or (3) a reference to an object of type T created during a prior field initialization; this sytematically treats aliasing. The HeapChoiceGenerator is used to gener- ate the choices. [25]

(Native Peers maybe not necessary) Most notably, SPF incorporates native peers that capture the calls to the java.lang.Math libraries and dispatch them to an appropriate constraint solver that can handle complex Math constraints. The same mechanism is also used for capturing String operations. [25]

(Listeners) The listeners gather and display information about the path conditions generated during the symbolic execution. They generate test cases and sequences in various user-defined formats. [25]

(Current limitations) Work in progress includes updating the symbolic string analysis and careful testing of the code citeLuckow2014. (However, mention how the current version has some limited support.)

3 Evaluation

4 Future Work

Bibliography

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

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

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

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

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

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

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

[8] 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[26]   Păsăreanu, C. S. and Visser, W. "Symbolic Execution and Model Checking for Testing". In: *HVC 2007* 4424.2007 (446), pp. 17–18.

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

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

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

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

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

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

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

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

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

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

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

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

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

5 Declaration of Academic Integrity

Thesis Statement pursuant to § 22 paragraph 7 of APB TU Darmstadt

I herewith formally declare that I have written the submitted thesis independently. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I clearly marked and separately listed all of the literature and all of the other sources which I employed when producing this academic work, either literally or in content. This thesis has not been handed in or published before in the same or similar form.

In the submitted thesis the written copies and the electronic version are identical in content.

Date: Signature:

List of Figures

List of Tables

List of Listings

Glossary

Lineage   Lineage description

List of Abbreviations and Acronyms

API   Application Programming Interface
CFG   Control Flow Graph
HDFS Hadoop Distributed File System
JPF   Java PathFinder
JVM   Java Virtual Machine
RDD   Resilient Distributed Dataset
SPF   Symbolic PathFinder