Symbolic Execution of Higher-order Functions in Big Data Systems

Omar Erminy

Introduction

Problem

Big data applications have become crucial in several areas, however, program verification techniques are seldom adapted to their context.

Introduction

TECHNISCHE UNIVERSITÄT DARMSTADT

Research Questions

1. Is symbolic execution suitable for Spark applications?

2. What are its particular characteristics?

3. Is there a symbolic execution framework that can be adapted?

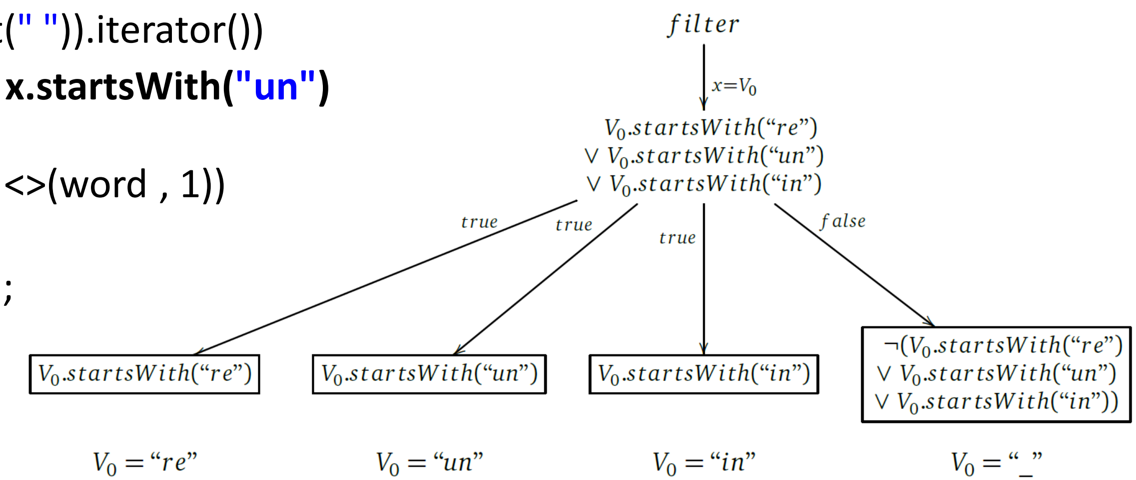
4. If so, what are its advantages and disadvantages?

Symbolic Execution of Spark Programs

Example – Extended Word Count

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

Symbolic Execution Tree



Minimal Input Dataset = {"*re un in _*"}

Example adapted from [3]

Background

Big Data Framework

Analysis Framework

Java PathFinder

References [1, 5]

Apache Spark

It is a **distributed large-scale** data processing framework.

It makes use of a **shared memory** abstraction called Resilient Distributed Dataset (**RDD**).

Many of the operations defined on its API are defined as **higher-order functions**.

References [1, 10]

Java PathFinder (JPF)

It is an **execution environment** for **verification** and analysis of Java **bytecode** programs.

Its default mode of operation is **explicit state model checking**.

References [4, 5, 6]

Java PathFinder (JPF)

Example – Random Numbers

```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);
  }
}
```

ArithmeticException: division by zero
For values:
a = 0 and b = 2
a = 1 and b = 1

Example taken from [5]

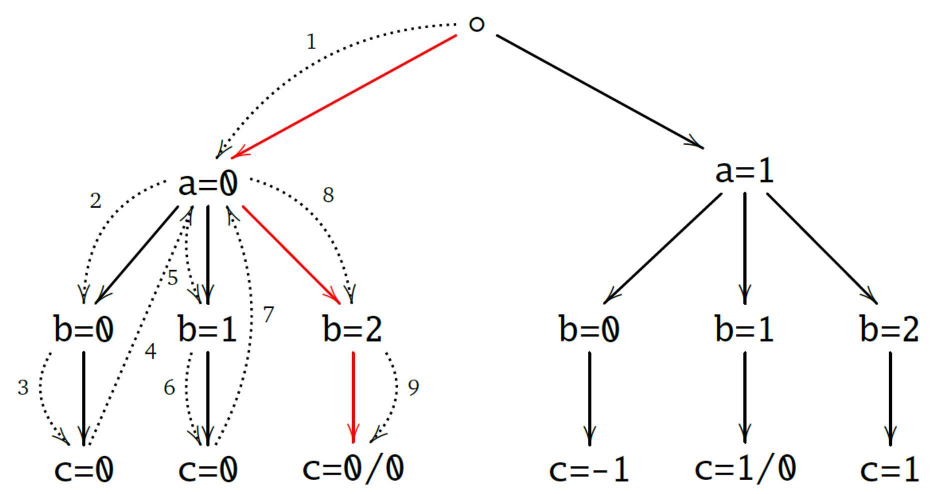
Java PathFinder (JPF)

TECHNISCHE UNIVERSITÄT DARMSTADT

Example – Random Numbers

```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);
  }
}
```



Example taken from [5]

Java PathFinder (JPF)

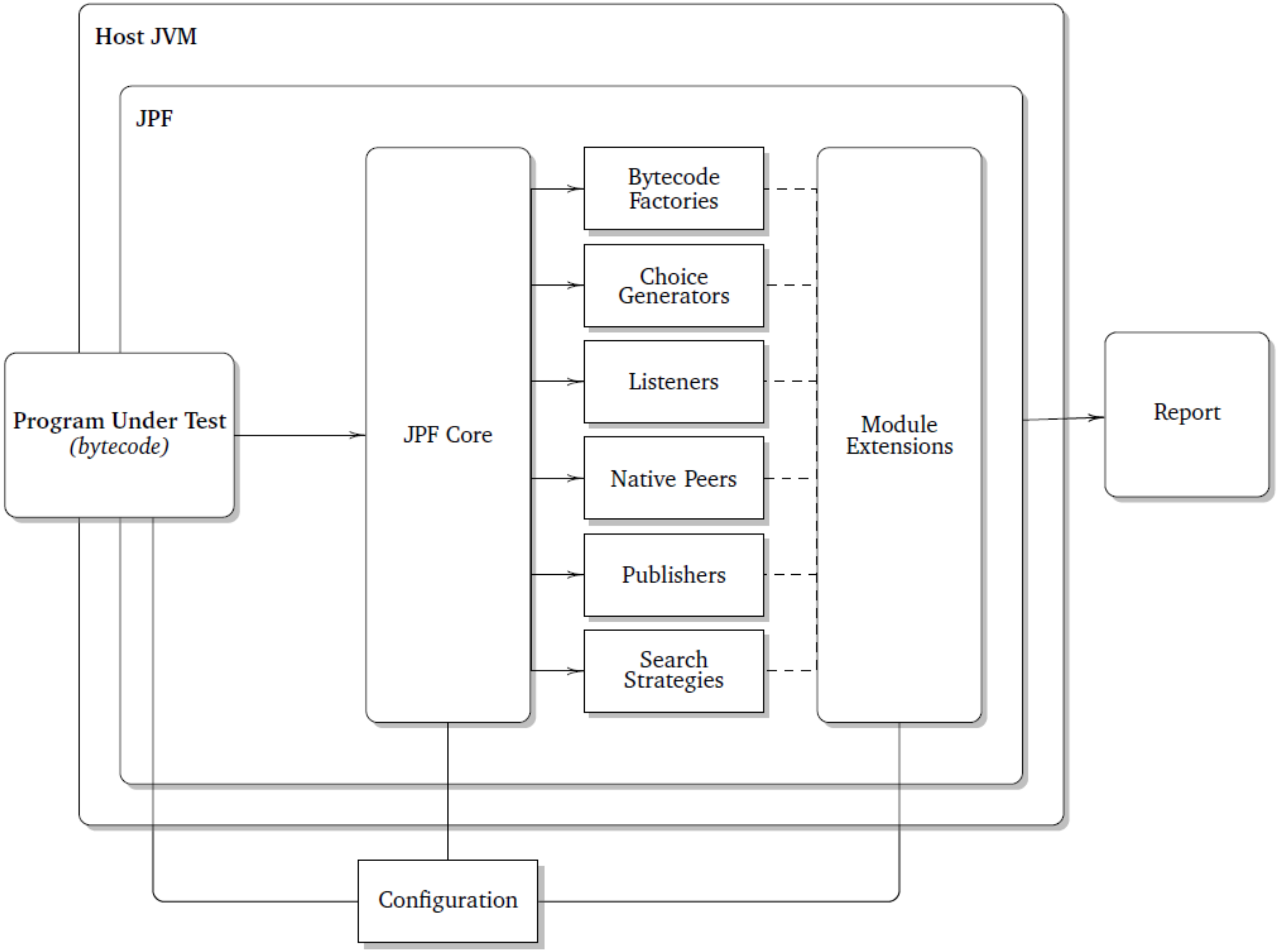
TECHNISCHE UNIVERSITÄT DARMSTADT

Components

- **Bytecode Factories**: Semantics
- **Choice Generators**: Checkpoints and state exploration
- **Listeners**: Property validation
- Native Peers
- Publishers
- Search Strategies

References [4, 5]

Java PathFinder (JPF)



Example adapted from [5]

Symbolic PathFinder (SPF)

Extensions

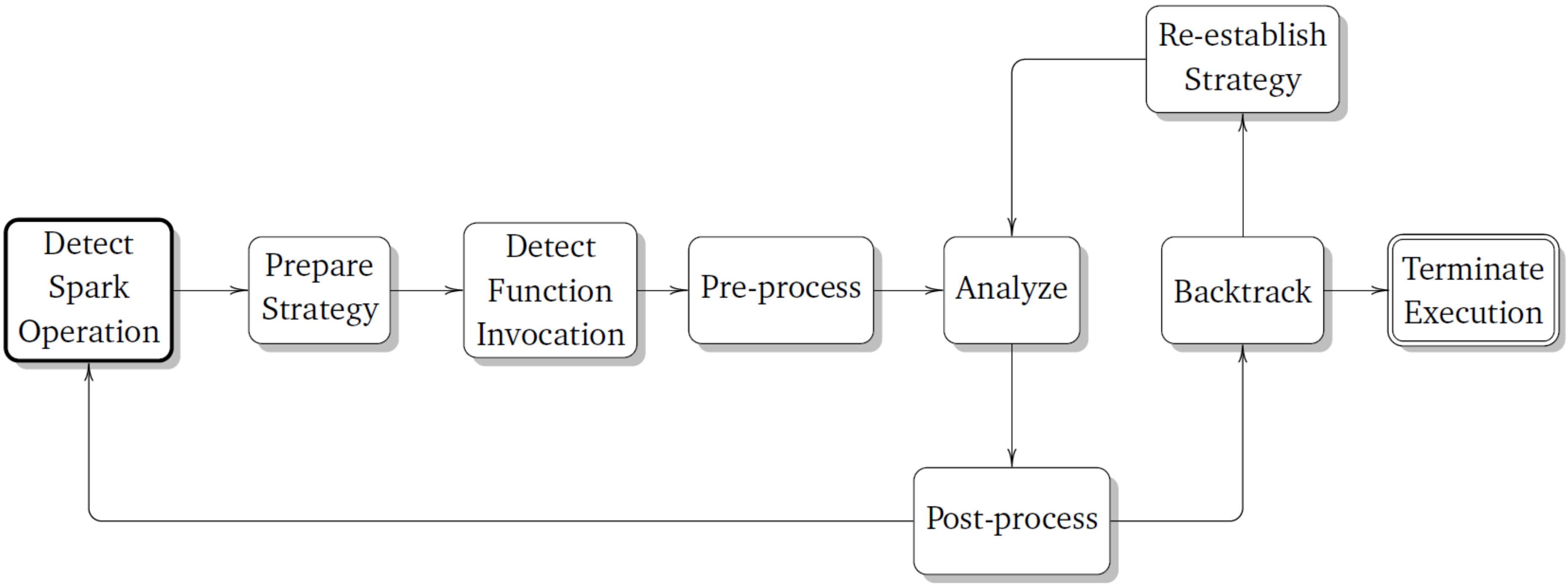
- **Bytecode Factory**: Full implementation of the symbolic semantics
- **Choice Generators**: Registered whenever a branching instruction is executed (*PCChoiceGenerator*)
- **Solvers**: Third-party constraint solvers used to process the path conditions (*Choco, Coral, CVC3*)

References [2, 7, 8, 9]

Symbolic Execution of Spark Programs

TECHNISCHE UNIVERSITÄT DARMSTADT

How to do it? – Conceptual Process

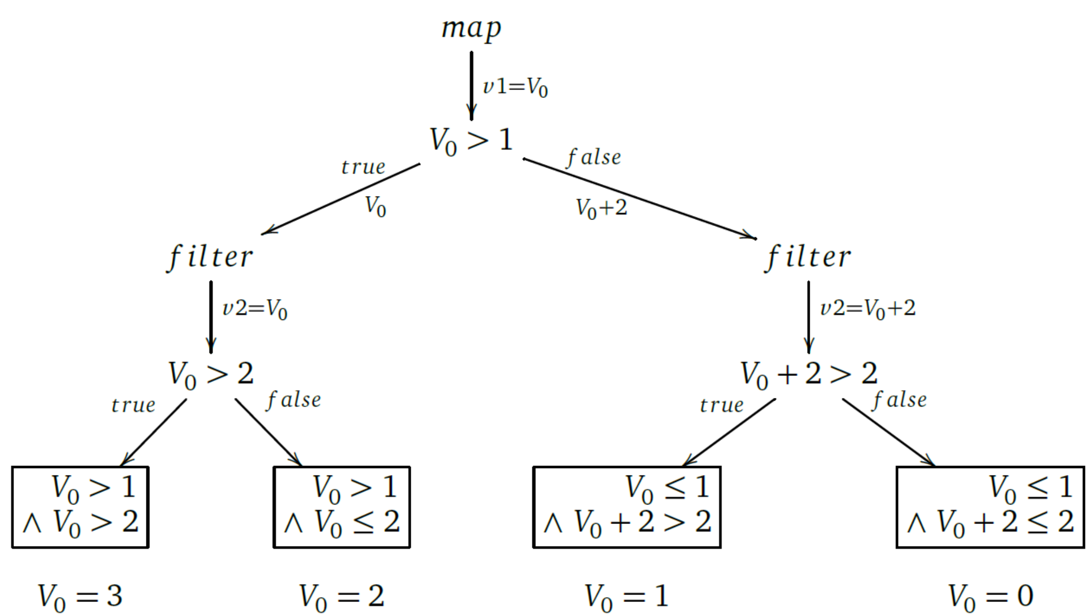


Symbolic Execution of Spark Programs

Example – Multiple Operations

```
numbers.map(v1 -> {
  if(v1 > 1) return v1;
  else return v1+2;
})
.filter(v2 -> v2 > 2);
```

Symbolic Execution Tree



Minimal Input Dataset = {3, 2, 1, 0}

Symbolic Execution of Spark Programs

TECHNISCHE UNIVERSITÄT DARMSTADT

How to do it? – JPF-SymSpark

JPF-SymSpark is a **JPF module built on top of SPF** whose goal is to coordinate the symbolic execution of Apache Spark programs.

Programs must be written for the **Java's RDD API**.

It produces a reduced input dataset that **ensures full path coverage**.

JPF-SymSpark

TECHNISCHE UNIVERSITÄT DARMSTADT

Extensions

- **Bytecode Factory**: Detection of Spark operations. Only interested in the *INVOKEVIRTUAL* instruction
- **Listeners**: Stateless orchestrator of the symbolic execution of a sequence of Spark operations
- **Choice Generators**: Used to correctly reproduce the behavior of some of the Spark operations
- **Publishers**: Produce a reduced input dataset and notify unfeasible path conditions

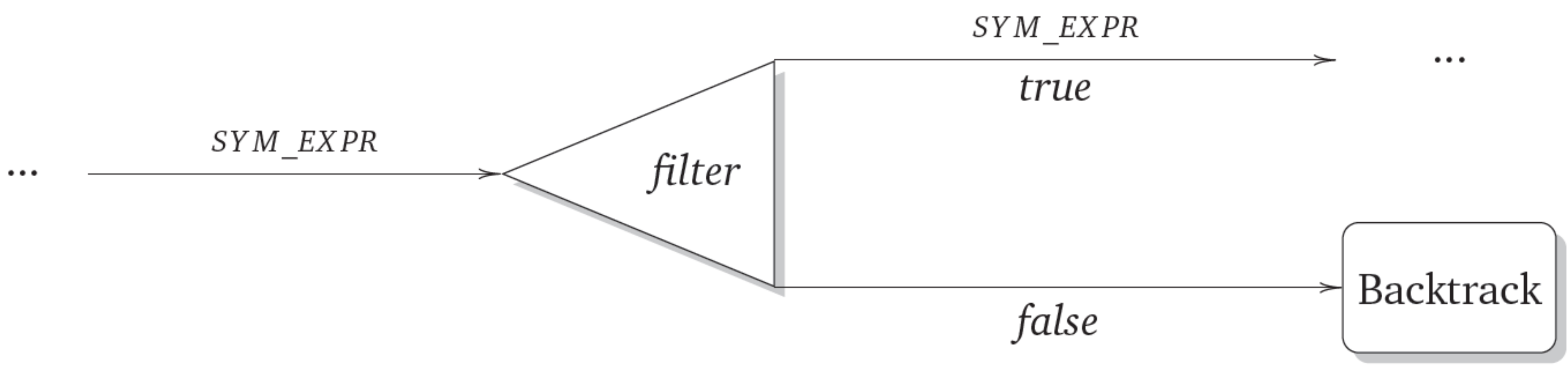
JPF-SymSpark

Components

- **Surrogate Spark Library**: Minimize external dependencies and native calls. Replace irrelevant implementations with simplified versions.
- **Method Coordinator**: Select the adequate strategy and keeps a global state of the analysis.
- **Method Strategies**: Define how operations deal with their symbolic input and output parameters

JPF-SymSpark - Strategies

Filter Strategy

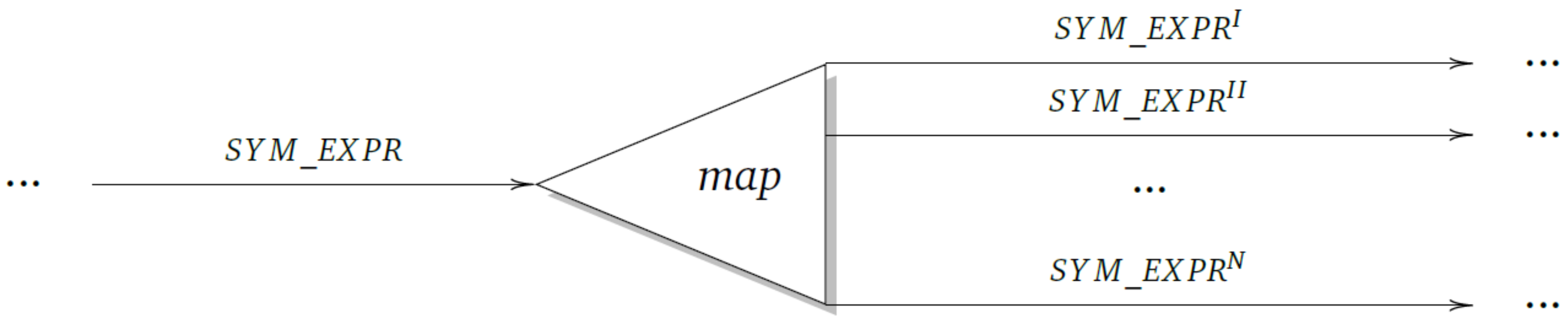


Pre-processing: Set input to symbolic expression passed by the coordinator.

Post-processing: Pass the same expression if *true*, backtrack if *false.*

JPF-SymSpark - Strategies

Map Strategy

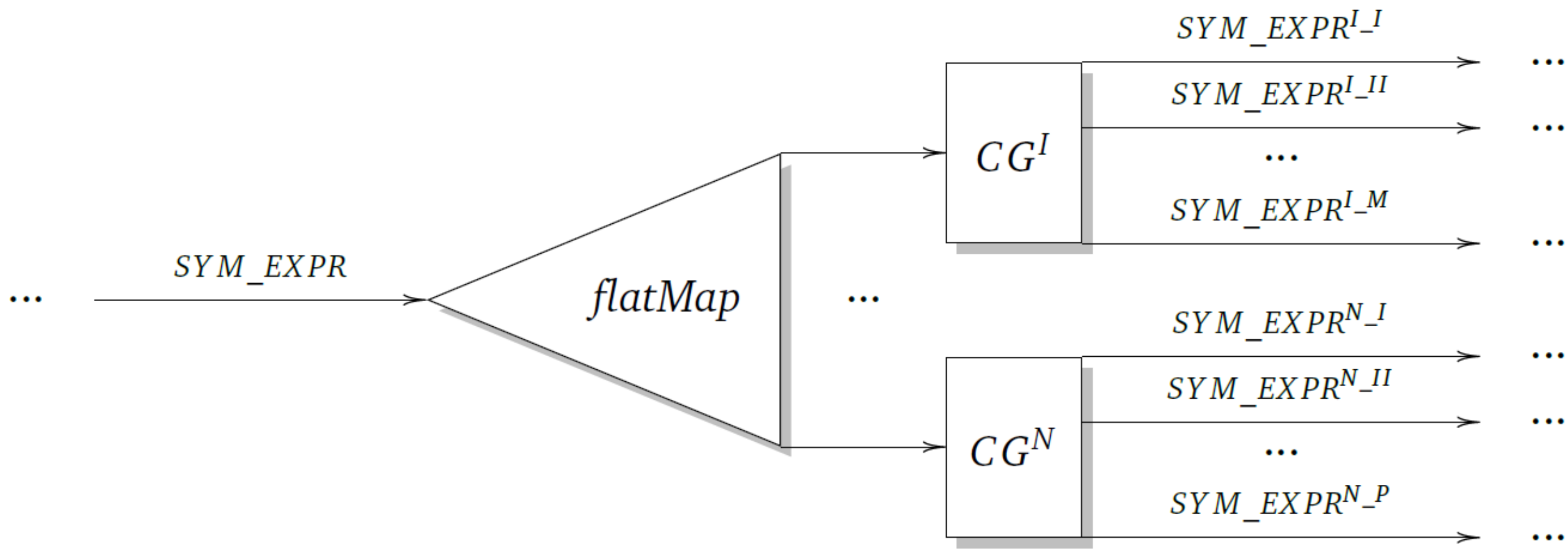


Pre-processing: Set input to symbolic expression passed by the coordinator.

Post-processing: Pass the symbolic expression resulting from the application of the function to the coordinator.

JPF-SymSpark - Strategies

FlatMap Strategy



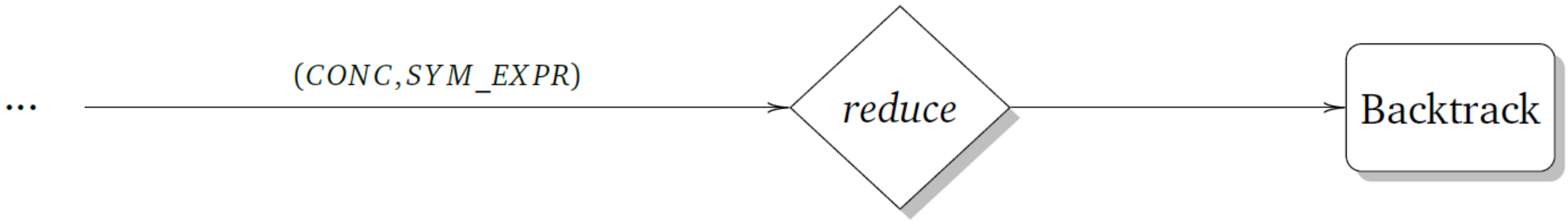
Pre-processing: Set input to symbolic expression passed by the coordinator.

Post-processing: Register a *SparkMultipleOutputChoiceGenerator* with all different symbolic expressions representing the different manipulations of the input value.

JPF-SymSpark - Strategies

Reduce Strategy

1) Non-Iterative



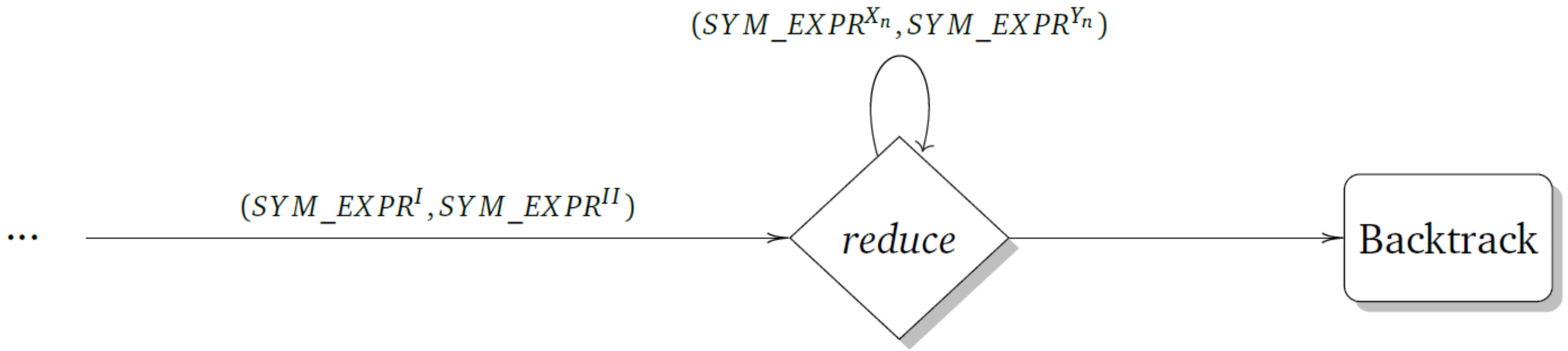
Pre-processing: Set single input to symbolic expression passed by the coordinator and the accumulated input as a concrete value.

Post-processing: Pass the same expression if *true*, backtrack if *false*

JPF-SymSpark - Strategies

Reduce Strategy

2) Iterative



Pre-processing: Register a *SparkIterativeChoiceGenerator* used to count the iterations. Set the accumulated and input parameters based on the iteration.

Post-processing: Update the choice generator with the output of each iteration and the update their respective path conditions.

Evaluation

Qualitative

Quantitative

References [1, 5]

Qualitative Evaluation

Requirements

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

Partially fulfilled. Support on symbolic String operations is constrained by the limitations of SPF.

Qualitative Evaluation

Requirements

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

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.

Qualitative Evaluation

TECHNISCHE UNIVERSITÄT DARMSTADT

Requirements

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

Partially fulfilled. This surrogate library is not exhaustive, for this reason, there will be unsupported operations that compile under the regular Spark library. Other Spark APIs are not supported.

Qualitative Evaluation

Requirements

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

Partially fulfilled. Only actions that work on primitive values are supported. The symbolic output of aggregate functions is not percolated beyond the boundaries of the operation.

Qualitative Evaluation

TECHNISCHE UNIVERSITÄT DARMSTADT

Summary

	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 †

- Partially or not met requirements limit the applicability of the tool.
- Most of the unfulfilled requirements result from limitations of SPF.
- The proposed process serves as a starting point for further research.

Quantitative Evaluation

Setup

- **Iterative Reduce with Non-Cumulative Condition (IRNC)**:
Single *reduce* action with a conditional defined on its non cumulative symbolic variable.

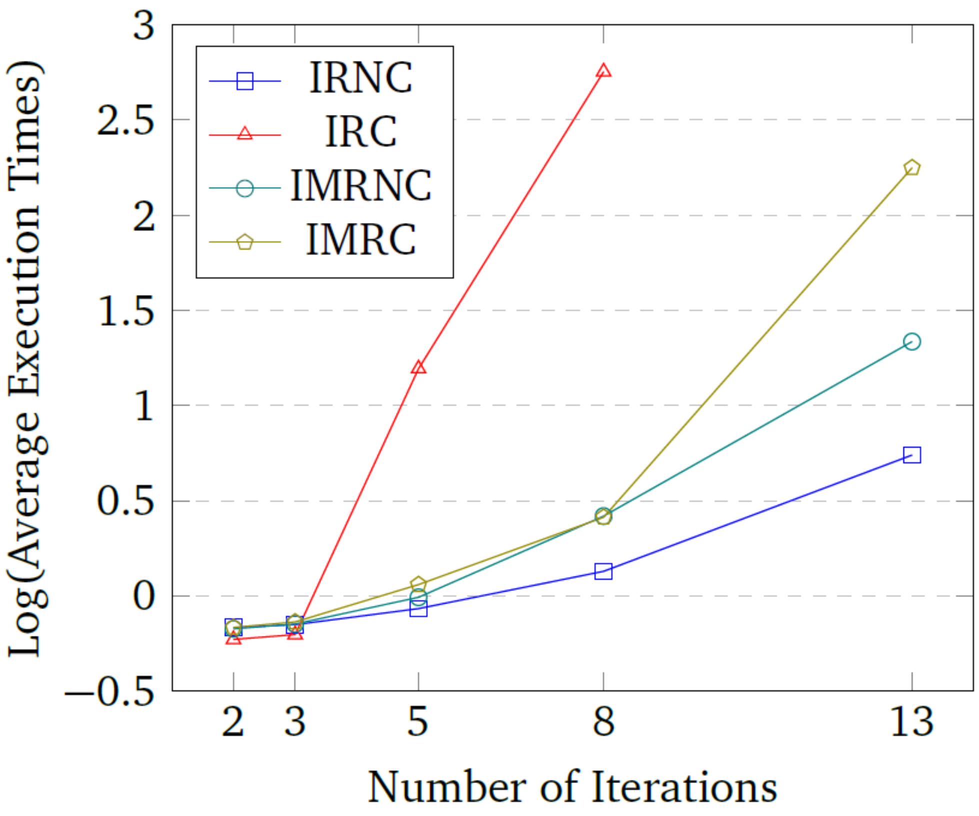
- **Iterative Reduce with Cumulative Condition (IRC)**:
Single *reduce* action with a conditional defined on its cumulative symbolic variable.

- **Iterative Map and Reduce with Non-Cumulative Condition (IMRNC)**:
A *map* and a *reduce* action with a conditional on its non cumulative symbolical variable.

- **Iterative Map and Reduce with Cumulative Condition (IMRC)**:
A *map* and a *reduce* action with a conditional on its cumulative symbolic variable.

Quantitative Evaluation

Execution Times



Quantitative Evaluation

TECHNISCHE UNIVERSITÄT DARMSTADT

Number of Path Conditions

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

Satisfiable (s), Unsatisfiable (u)

Conclusion

Research Questions Revisited

1. *Is symbolic execution a suitable for Spark applications?*

Yes. As *JPF-SymSpark* demonstrates, the technique can be used to analyze Apache Spark programs

Conclusion

TECHNISCHE UNIVERSITÄT DARMSTADT

Research Questions Revisited

2. What are the particular characteristics?

Two characteristics: Control flow instructions of an application can be contained inside the functions and some Spark operations have control flow semantics defined intrinsically.

Conclusion

Research Questions Revisited

3. *Is there a symbolic execution framework that can be adapted?*

Symbolic PathFinder is the most complete framework available and with the most amount of documentation.

Conclusion

TECHNISCHE UNIVERSITÄT DARMSTADT

Research Questions Revisited

4. If so, what are its advantages and disadvantages?

- Powerful tool with full symbolic semantics for most of the primitive types.
- Convenient management of path conditions
- Limited support of symbolic Strings and data structures
- Sporadic new releases and decaying codebase

Conclusion

- The lack of support of symbolic data structures and partial support of symbolic String operations pose as major limitations.
- Cumulative symbolic variables translate into more complex path conditions resulting in poor time performance.
- Higher numbers of unsatisfiable path conditions allow a faster exploration of the state space.
- Symbolic constraint solvers pose as the main bottlenecks.

References

1. *Apache Spark™ - Lightning-Fast Cluster Computing*. U R L: http://spark.apache.org/ (visited on 2017).

2. 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. I S B N: 978-3-540-73367-6.

3. Dean, J. and Ghemawat, S. “MapReduce: Simplified Data Processing on Large Clusters”. In: *Communications of the ACM* 51.1 (Jan. 2008), pp. 107–113. I S S N: 0001-0782. D O I: 10.1145/1327452.1327492.

4. 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. I S S N: 1433-2779. D O I: 10.1007/s100090050043.

5. *Java PathFinder*. National Aeronautics and Space Administration. U R L: http://babelfish.arc.nasa.gov/trac/jpf/wiki (visited on 2017).

6. *NASA’s Ames Research Center*. National Aeronautics and Space Administration. U R L: https://www.nasa.gov/centers/ames/home/index.html (visited on 2017).

7. 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. I S B N: 978-1-4503-0116-9. D O I: 10.1145/1858996.1859035.

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

References

9. 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. I S B N: 978-3-642-20398-5. D O I: 10.1007/978-3-642-20398-5_26.

10. 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. I S S N: 00221112. D O I: 10.1111/j.1095-8649.2005.00662.x.

Questions