Temporal Random Testing for Spark Streaming

A. Riesco and J. Rodríguez-Hortalá

Universidad Complutense de Madrid, Madrid, Spain

ScalaMAD @ Indizen
January 13th, 2015

Spark

- The core of Spark is a batch computing framework based on manipulating so called *Resilient Distributed Datasets* (RDDs).
- RDDs provide a fault tolerant implementation of distributed immutable multisets.
- Computations are defined as transformations on RDDs.
- The set of predefined RDD transformations includes typical higher-order functions from functional programming like map, filter, etc.
- It also includes aggregations by key and joins for RDDs of key-value pairs.
- We can also use Spark actions, which allow us to collect results into the program driver, or store them into an external data store.

Spark

- The following example uses the Scala Spark shell to implement a variant of the word count example that computes the number of occurrences of each character:

```
scala> val cs = sc.parallelize("let's count some letters", numSlices=4)
cs: org.apache.spark.rdd.RDD[Char] = ParallelCollectionRDD[5] at parallelize at <console>:21
scala> cs.map{(_, 1)}.reduceByKey{_+_}.collect()
15/12/08 21:19:24 INFO SparkContext: Starting job: collect at <console>:24
15/12/08 21:19:24 INFO DAGScheduler: Registering RDD 7 (map at <console>:24)
...
15/12/08 21:19:24 INFO DAGScheduler: Job 2 finished: collect at <console>:24, took 0.027890 s
res4: Array[(Char, Int)] = Array((t,4), ( ,3), (l,2), (e,4), (u,1), (m,1), (n,1), (r,1),
                                 (',1), (s,3), (o,2), (c,1))
```

Spark Streaming

- These notions of transformations and actions are extended in Spark Streaming from RDDs to *DStreams* (Discretized Streams).
- DStreams are series of RDDs corresponding to micro-batches.
- These batches are generated at a fixed rate according to the configured *batch interval*.
- Spark Streaming is synchronous: given a collection of input and transformed DStreams, all the batches for each DStream are generated at the same time as the batch interval is met.
- Actions on DStreams are also periodic and are executed synchronously for each micro batch.

Spark Streaming

- The letter count in Spark Streaming is implemented as follows:

```
object HelloSparkStreaming extends App {
  val conf = new SparkConf().setAppName("HelloSparkStreaming")
                            .setMaster("local[5]")
  val sc = new SparkContext(conf)
  val batchInterval = Duration(100)
  val ssc = new StreamingContext(sc, batchInterval)

  val batches = "let's count some letters, again and again"
                .grouped(4)
  val queue = new Queue[RDD[Char]]
  queue ++= batches.map(sc.parallelize(_, numSlices = 4))
  val css = ssc.queueStream(queue, oneAtATime = true)
  css.map{(_, 1)}.reduceByKey{_+_}.print()
  ssc.start()
  ssc.awaitTerminationOrTimeout(5000)
  ssc.stop(stopSparkContext = true)
}
```

```
-------------------------
Time: 1449638784400 ms
-------------------------
(e,1)
(t,1)
(l,1)
(',1)
...
-------------------------
Time: 1449638785300 ms
-------------------------
(i,1)
(a,2)
(g,1)
...
-------------------------
Time: 1449638785400 ms
-------------------------
(n,1)
...
```

Spark Streaming

- We have the following DS, where each box is a batch containing a `String`

  `let'` `s co` `...` `agai` `n`

- It counts the letters for each RDD, so we have an output stream:

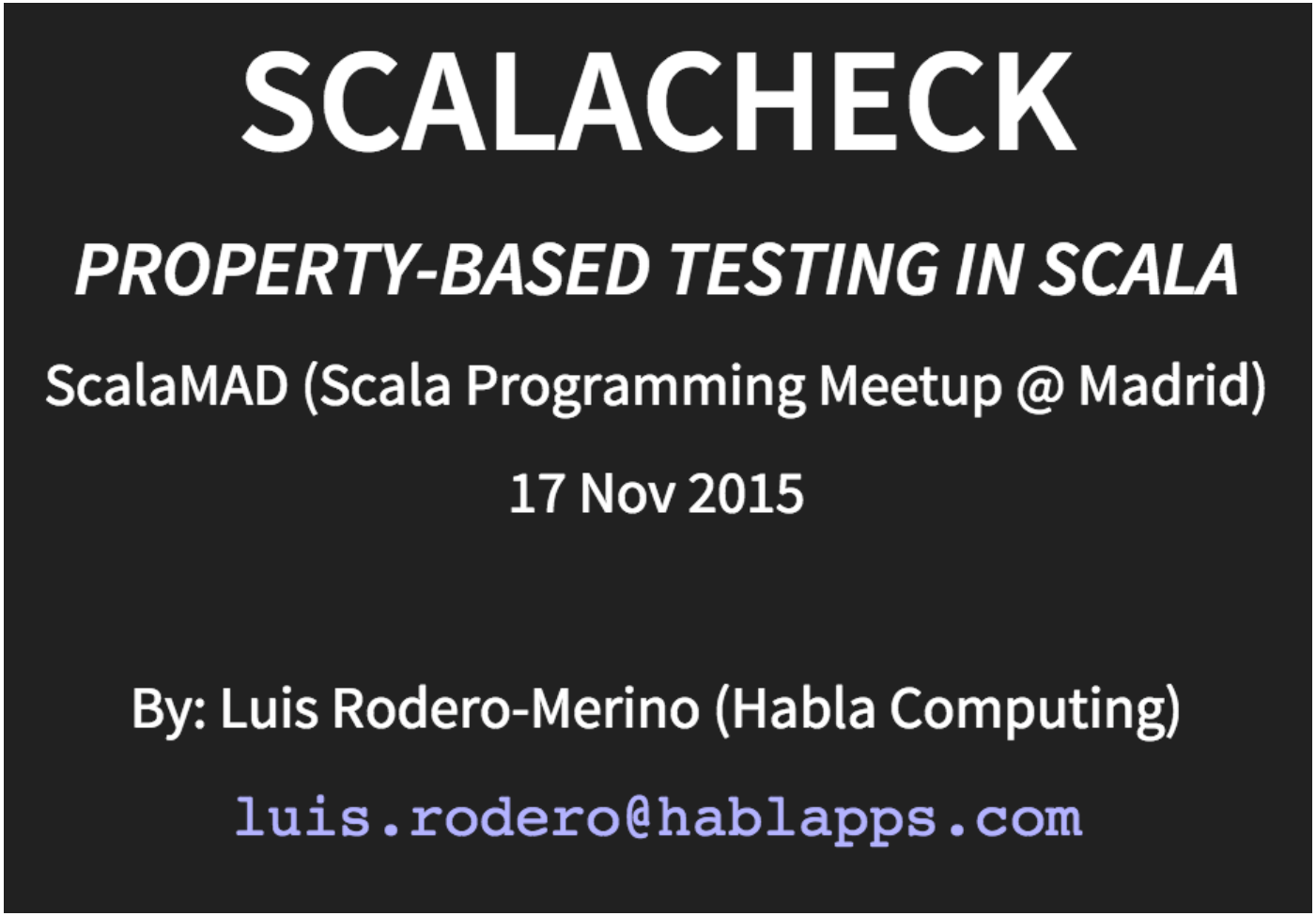
  `(e,1),(t,1),(l,1),(',1)` `( ,1),(c,1),(o,1),(s,1)` `...`

  `(i,1),(a,2),(g,1)` `(n,1)`

Property-based testing

- In *Property-based testing* tests are stated as properties, which are first order logic formulas that relate program inputs and outputs.
- PBT is based on *generators* and *properties*, that together with a quantifier form a property.
  - Generators are non-deterministic functions that produce random values for the inputs, aka test cases.
  - Simple formulas are defined as assertions on the generated test cases.
  - Formulas can be universally or existentially quantified.
  - Properties can be composed by using logical operators.
- The main advantage is that the assertions are exercised against hundreds of generated test cases, instead of against a single value like in xUnit frameworks

Property-based testing in Scala: ScalaCheck



http://lrodero.github.io/Intro-to-ScalaCheck_Slides-for-ScalaMAD/

Property-based testing in Scala: ScalaCheck

PROPERTY-BASED?

Usually we set the expected output for some given input

```
// Specs2 example
class HelloWorldSpec extends Specification {
  (...)
  def e1 = "Hello world" must haveSize(11)
}
```

A **property** is set for *any* input

```
val prSqrt: Prop = forAll{(n: Int) => scala.math.sqrt(n*n) == n}
val notPrSqrt: Prop = exists{(n: Int) => scala.math.sqrt(n*n) != n}
```

"(...) ScalaCheck can be used to state props about isolated parts - units - of your code (usually methods)"

Mostly, this refers to testing pure functions, *i.e.* with no side-effects, especially changes on state.

Property-based testing in Scala: ScalaCheck

CHECKING PROPERTIES

```
val concatStrPr: Prop = forAll {(s1:String, s2:String) =>
  val concat = (s1+s2);
  concat.endsWith(s2) &&
  concat.startsWith(s1) &&
  concat.size == s1.size + s2.size
}
concatStrPr.check // <-- cheching!
```

```
val sumListPr: Prop = forAll {(l:List[Int]) =>
  (l.sum > 0) ==> (l.indexWhere(_ > 0) > 0)
}
sumListPr.check
```

Property-based testing in Scala: ScalaCheck

GENERATORS

Create random values (of any kind)

Used by ScalaCheck to generate test cases

Can be used in any other application or testing environment

```
class Gen[+T]{
  (...)
  def sample: Option[T]
  (...)
}
```

Property-based testing in Scala: ScalaCheck

PROVIDED GENS - BASIC

```
val g1: Gen[Int] = Gen.choose(0,10)
val g2: Gen[Double] = Gen.choose(0,0.5)
val g3: Gen[Int] = Gen.chooseNum(-10,10)
val g4: Gen[Double] = Gen.posNum[Double]
```

```
Gen.alphaStr: Gen[String]; Gen.numStr: Gen[String]
Gen.identifier: Gen[String]; Gen.uuid: Gen[java.util.UUID]
```

```
val g5: Gen[String] = Gen.oneOf("Meat","Vegs")
val g6: Gen[String] = Gen.frequency((4,"Meat"),(1,"Vegs"))
val g7: Gen[Seq[String]] = Gen.someOf("Apple", "Orange", "Banana")

// Applicable also with other gens
val g8: Gen[Int] = Gen.frequency((4,Gen.choose(0,10)),
                                 (1,Gen.choose(10,100000)))
```

ScalaCheck for core Spark

- Applying ScalaCheck for testing programs that manipulate Spark RDDs is quite easy.
- The main difficulty is ensuring that the Spark context is shared by all the test cases.
- If we are going to generate around 100 test cases per ScalaCheck property, creating a new Spark context per test case would not be practical.
- Besides, we cannot have more than a Spark context running on the same JVM.
- Using the integration with Specs2, and Specs2's `BeforeAfterAll` trait leads to an easier and more robust solution.

ScalaCheck for core Spark

- This trait provides a method `sc` that can be used to parallelize lists generated with the built-in ScalaCheck generators.
- The Spark master or the parallelism level (default number of Spark partitions used to parallelize sequences) can also be customized by overriding the corresponding method.
- That Spark context is also available as an implicit value, that can be then used with the implicit conversions and generator provided by the object `RDDGen`.
- `RDDGen` objects are basically shortcuts to parallelize lists generated by built-in ScalaCheck generators.

ScalaCheck for core Spark

```
object RDDGen {
  /** Convert a ScalaCheck generator of Seq into a generator of RDD
   * */
  implicit def seqGen2RDDGen[A](sg : Gen[Seq[A]])
              (implicit aCt: ClassTag[A], sc : SparkContext,
              parallelism : Parallelism) : Gen[RDD[A]] =
    sg.map(sc.parallelize(_, numSlices = parallelism.numSlices))

  /** Convert a sequence into a RDD
   * */
  implicit def seq2RDD[A](seq : Seq[A])(implicit aCt: ClassTag[A],
                    sc : SparkContext, parallelism : Parallelism) : RDD[A]
    sc.parallelize(seq, numSlices=parallelism.numSlices)
```

ScalaCheck for core Spark

```
/** @returns a generator of RDD that generates its elements from g
 * */
def of[A](g : => Gen[A])
         (implicit aCt: ClassTag[A], sc : SparkContext,
          parallelism : Parallelism) : Gen[RDD[A]] =
  // this way is much simpler that implementing this with a
  // ScalaCheck Buildable, because that implies defining a
  // wrapper to convert RDD into Traversable
  seqGen2RDDGen(Gen.listOf(g))

/** @returns a generator of RDD that generates its elements from g
* */
def ofN[A](n : Int, g : Gen[A])
   (implicit aCt: ClassTag[A], sc : SparkContext,
    parallelism : Parallelism) : Gen[RDD[A]] = {
  seqGen2RDDGen(Gen.listOfN(n, g))
}
```

ScalaCheck for core Spark

```
   /** @returns a generator of RDD that generates its elements from g
  * */
  def ofNtoM[A](n : Int, m : Int, g : => Gen[A])
        (implicit aCt: ClassTag[A], sc : SparkContext,
         parallelism : Parallelism) : Gen[RDD[A]] =
    seqGen2RDDGen(UtilsGen.containerOfNtoM[List, A](n, m, g))
}
```

Property-based testing for Spark Streaming?

- "Adapting" ScalaCheck for core Spark was easy.
- Why not trying the same approach with Spark Streaming?

Property-based testing for Spark Streaming?

Problem 1

- Besides a Spark context, Spark Streaming programs use a streaming context to define the computations, which are transformations or periodic actions executed over DStreams.
- Each streaming context is associated to a Spark context, and it is a lightweight object that can be created quickly, but it has to be started and stopped explicitly, not only created.
- Besides, all the transformations and actions on DStreams have to be defined before the streaming context has started.
- Finally, only a single streaming context can be active per JVM, and streaming contexts cannot be restarted.
- This is a complex lifecycle that needs to be handled with care.

Property-based testing for Spark Streaming?

Problem 2

- Spark Streaming programs are designed to run forever.
- We need to a way to determine when all the assertions relevant to a test case have been executed completely, so we can then finish the test case by stopping the streaming context.

Property-based testing for Spark Streaming?

Problem 3

- DStream batches are generated with a fixed frequency that we call the "batch interval."
- All the batches are expected to be completed at the same speed but, as we are generating random test cases, then we will often have batches significantly bigger than others, so in practice some batches will be computed faster than others.
- In general some input values are faster to compute than others (as an extreme example consider a transformation that computes the i-th prime number for each input number i).
- As the chosen batch interval must leave enough time to compute the slowest batches, this might lead to wasting time when computing the fastest batches, and tests not running as fast as they should.

Property-based testing for Spark Streaming?

Problem 4

- As DStreams are meant to run nonstop, the Spark Streaming runtime captures any exception generated when computing a batch, to prevent stopping the computation.

Property-based testing for Spark Streaming?

- We can reuse the same Spark context for several streaming contexts.
- That leads naturally to a test life cycle where we create and stop Spark contexts with `BeforeAfterAll`, and create and stop Spark streaming contexts with `BeforeAfterEach`.
- We still need to manually start the streaming context in the body of the test case, but only after declaring the derived DStreams defined by applying the test subject, and the actions that apply the assertions that characterize the test.

Property-based testing for Spark Streaming?

- To avoid losing the results of the matchers, due to Spark captured exceptions, we can use Specs2's Result type, starting from `ok` and combining the result obtained for each batch with Specs2's `and` operator.
- Finally, in order to determine when the test can finish, we can register a `StreamingListener` in the streaming context, that notifies a `SyncVar` each time a batch is completed.
- We can use the `SyncVar` to block waiting for completion of a fixed number of batches: this is implemented in the method awaitForNBatchesCompleted of the object `StreamingContextUtils`.
- That is all we need to define simple unit test for Spark Streaming.

Property-based testing for Spark Streaming?

```
def successfulSimpleQueueStreamTest =
          simpleQueueStreamTest(expectedCount = 0)
  def failingSimpleQueueStreamTest =
          simpleQueueStreamTest(expectedCount = 1) must beFailing

  def simpleQueueStreamTest(expectedCount : Int) : Result = {
    val record = "hola"
    val batches = Seq.fill(5)(Seq.fill(10)(record))
    val queue = new Queue[RDD[String]]
    queue ++= batches.map(batch =>
           sc.parallelize(batch, numSlices = defaultParallelism))
    val inputDStream = ssc.queueStream(queue, oneAtATime = true)
    val sizesDStream = inputDStream.map(_.length)
```

Property-based testing for Spark Streaming?

```
var batchCount = 0
var result : Result = ok
inputDStream.foreachRDD { rdd =>
  batchCount += 1
  println(s"completed batch number $batchCount:
                                    ${rdd.collect.mkString(",")}")
  result = result and {
    rdd.filter(_!= record).count() === expectedCount
    rdd should existsRecord(_ == "hola")
  }
}
sizesDStream.foreachRDD { rdd =>
  result = result and {
    rdd should foreachRecord(record.length)(len => _ == len)
  }
}
```

Property-based testing for Spark Streaming?

```
    // should only start the dstream after all the transformations
    // and actions have been defined
    ssc.start()

    // wait for completion of batches.length batches
    StreamingContextUtils.awaitForNBatchesCompleted
                          (batches.length, atMost = 10 seconds)(ssc)

    result
  }
```

Property-based testing for Spark Streaming?

- DStreams are meant to run forever, while a test case should be executed in a finite time: what is a test case?
- How are generators?
- There can be relations between the values in a DStream that cannot be expressed in first order logic.
- We can use temporal logic.

Property-based testing for Spark Streaming

```
class StreamingFormulaDemo1 {

  def faultyCount(ds : DStream[Double]) : DStream[Long] =
    ds.count.transform(_.map(_ - 1))
  def countForallAlwaysProp(testSubject : DStream[Double] =>
                                          DStream[Long]) = {
    type U = (RDD[Double], RDD[Long])
    val (inBatch, transBatch) = ((_ : U)._1, (_ : U)._2)
    val numBatches = 10
    val formula : Formula[U] = always { (u : U) =>
      transBatch(u).count === 1 and
      inBatch(u).count === transBatch(u).first
    } during numBatches
    val gen = BatchGen.always(BatchGen.ofNtoM(10, 50,
                                arbitrary[Double]), numBatches)
    forAllDStream(gen)(testSubject)(formula)
  }.set(minTestsOk = 10).verbose  }
```

Property-based testing for Spark Streaming

- We would like to test a Spark Streaming program that takes a stream of user activity data and returns a stream of banned users.
- To keep it simple we assume that the input records are pairs containing a `Long` user id, and a `Boolean` indicating whether the user has been honest at that instant.
- So, the test subject that implement this has type `testSubject : DStream[(Long, Boolean)] => DStream[Long]`.

Property-based testing for Spark Streaming

- Assume now that we implement something like this:

```
def statelessListBannedUsers(ds : DStream[(Long, Boolean)]) :
        DStream[Long] = ds.map(_._1)
```

- Is it possible to define generators and properties to test it?
- We would need our own generators.

Property-based testing for Spark Streaming

- Generators at batch level are similar to those in Scalacheck:

```
val batchSize = 20
val (badId, ids) = (15L, Gen.choose(1L, 50L))
val goodBatch = BatchGen.ofN(batchSize, ids.map((_, true)))
val badBatch = goodBatch + BatchGen.ofN(1, (badId, false))
```

Property-based testing for Spark Streaming

- Similarly, properties at batch level can be defined as follows:

```
type U = (RDD[(Long, Boolean)], RDD[Long])
val (inBatch, outBatch) = ((_ : U)._1, (_ : U)._2)
  val allGoodInputs =
        at(inBatch)(_ should foreachRecord(_._2 == true))
  val badInput =
        at(inBatch)(_ should existsRecord(_ == (badId, false)))
  val noIdBanned = at(outBatch)(_.isEmpty)
  val badIdBanned = at(outBatch)(_ should existsRecord(_ == badId))
```

- However, these generators/properties do not represent the behavior of DStreams, since they are only concerned with specific "moments".
- Here `foreachRecord` and `existsRecord` are custom Specs2 matches that check a Boolean function for all or for any of the records in an RDD.
- `at()` simply composes a projection with a function that returns a Specs2 matcher, and it is used to define assertions on the batch in the current instant

LTL_{ss}

- We can use temporal logic where *time* does not refer to "wall-clock time" but to logic, discrete time synched with the batch interval.
- The operators that can be used in this kind of logic are:

  Always, which indicates that something always happens.

  Eventually, which indicates that something happens in some moment in the future.

  Until, which indicates that something happens for some time and, before it stops happening, another action happens.

  Next, which indicates that something happens in the next instant.

LTL_{ss}

- We can state properties like
  - *Always* `allGoodInputs` implies *never* `badIdBanned`.
  - *Always* `allGoodInputs` implies *always* `noIdBanned`.
  - *Eventually* (`badInput` implies *always* (from that point) `badIdBanned`).
  - `allGoodInputs` *until* (`badInput` implies *always* (from that point) `badIdBanned`).
- We could also use these operators to define the generators:
  - `allGoodInputs` *until* `badInput`.

LTL_{ss}

- The problem now is that *always* is a very long time.
- It is not possible in general to prove properties in this logic.
- In practice, DS are finite.
- Moreover, we are interested in the time it can take to achieve a result.
- Hence, we decided to extend the logic with timeouts.

LTL_{ss}

- The operators in the logic are:

  Always becomes *always for the next n batches*, and indicates that a property holds for the next n batches.

  Eventually becomes *eventually in the next n batches*, and indicates that a property holds in at least one of the next n batches.

  Until becomes φ_1 *until* φ_2 *in the next n batches*, and indicates that, before n batches have passed, φ_2 must hold and, for all batches before that, φ_1 must hold.

  Next... does not change.

Generators for Spark Streaming

- We can use temporal operators with timeout and composition of streams to define generators:

```
val (headTimeout, tailTimeout, nestedTimeout) = (10, 10, 5)
val gen = BatchGen.until(goodBatch, badBatch, headTimeout) ++
          BatchGen.always(Gen.oneOf(goodBatch, badBatch),
                          tailTimeout)
```

Properties for Spark Streaming

- Similarly, we can complete our formula as follows:

```
type U = (RDD[(Long, Boolean)], RDD[Long])
val (inBatch, outBatch) = ((_ : U)._1, (_ : U)._2)
val formula : Formula[U] = {
  val allGoodInputs = at(inBatch)(_ should foreachRecord(_._2 == true))
  val badInput = at(inBatch)(_ should existsRecord(_ == (badId, false)))
  val noIdBanned = at(outBatch)(_.isEmpty)
  val badIdBanned = at(outBatch)(_ should existsRecord(_ == badId))

  ((allGoodInputs and noIdBanned) until badIdBanned on headTimeout) and
  (always { badInput ==> (always(badIdBanned) during nestedTimeout) }
           during tailTimeout)
}
```

Properties for Spark Streaming: Twitter

Example

Upcoming features

- Random streams are expensive to generate.
- Moreover, since the generation is random, the streams revealing the errar might not be generated again.
- Hence, we are working to save those streams that do not fulfill the formula.
- We also plan to define generators that can read a file.
- This would allows us not just loading the previously saved files, but also load real data.
- Parallel execution of several test cases: the current implementation evaluates each test case in parallel (each batch and its actions are evaluated in parallel) but test cases are executed one after the other

Conclusions

- This framework is a prototype that has to be further tested and improved.
- Collaborating as a Databricks research partner.
- Any kind of collaboration is welcome!

`https://github.com/juanrh/sscheck/wiki/Quickstart`

`ariesco@fdi.ucm.es`