Temporal Random Testing for Spark Streaming

A. Riesco and J. Rodríguez-Hortalá

Universidad Complutense de Madrid, Madrid, Spain

12th International Conference on integrated Formal Methods, iFM 2016
June 3, 2016

Preliminaries

- With the rise of Big Data technologies, distributed stream processing systems (SPS) have gained popularity in the last years.
- These systems are used to continuously process high volume streams of data.
- Applications range from anomaly detection, low latency social media data aggregation, or the emergent IoT market.
- Although the first precedents of stream processing systems were developed in the 90s, with the boom of SPS a plethora of new systems have arisen.
- They are characterized by a distributed architecture designed for horizontal scaling.
- Among them Spark Streaming stands out as a particularly attractive option, with a growing adoption in the industry.

Spark

- Apache Spark is a fast and general engine for large-scale data processing.
- Programs are executed up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk.
- This performance is obtained thanks to its capabilities for in memory processing and caching for iterative algorithms.
- Spark provides a collection-based higher level API inspired in functional programming.

Spark

- It also presents a "batteries included" philosophy accelerates the development of Big Data processing applications.
- These "batteries" include libraries for scalable machine learning, graph processing, an SQL engine, and Spark Streaming.
- Spark programs can be written in Java, Scala, Python, or R.
- 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.

Spark

- The set of predefined RDD transformations includes typical higher-order functions 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.
- Actions are impure, so idempotent actions are recommended in order to ensure a deterministic behavior even in the presence of recomputations.

Spark

```
scala> val cs : RDD[Char] = sc.parallelize("let's count some
                                            letters", numSlices=3)
scala> cs.map{(_, 1)}.reduceByKey{_+_}.collect()
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))
```

1. By using `parallelize` we obtain an RDD {`let's count some letters`} with 3 partitions.
2. Applying `map` we have {`(l,1)(e,1)(t,1)(',1)(s,1)( ,1)(c,1)(o,1)(u,1)(n,1)(t,1)( ,1)(s,1)(o,1)...`}
3. The function `reduceByKey` applies addition to the second component of those pairs whose first component is the same.
4. The action `collect` allows us to print the final result.

Spark

```
scala> val cs : RDD[Char] = sc.parallelize("let's count some
                                            letters", numSlices=3)
scala> cs.map{(_, 1)}.reduceByKey{_+_}.collect()
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))
```

1. By using `parallelize` we obtain an RDD {`let's count some letters`} with 3 partitions.
2. Applying `map` we have {`(l,1)(e,1)(t,1)(',1)(s,1)( ,1)(c,1)(o,1)(u,1)(n,1)(t,1)( ,1)(s,1)(o,1)...`}
3. The function `reduceByKey` applies addition to the second component of those pairs whose first component is the same.
4. The action `collect` allows us to print the final result.

Spark

```
scala> val cs : RDD[Char] = sc.parallelize("let's count some
                                            letters", numSlices=3)
scala> cs.map{(_, 1)}.reduceByKey{_+_}.collect()
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))
```

1. By using `parallelize` we obtain an RDD {`let's count some letters`} with 3 partitions.
2. Applying `map` we have {`(l,1)(e,1)(t,1)(',1)(s,1)( ,1)(c,1)(o,1)(u,1)(n,1)(t,1)( ,1)(s,1)(o,1)...`}
3. The function `reduceByKey` applies addition to the second component of those pairs whose first component is the same.
4. The action `collect` allows us to print the final result.

Spark

```
scala> val cs : RDD[Char] = sc.parallelize("let's count some
                                            letters", numSlices=3)
scala> cs.map{(_, 1)}.reduceByKey{_+_}.collect()
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))
```

1. By using `parallelize` we obtain an RDD {`let's count some letters`} with 3 partitions.
2. Applying `map` we have {`(l,1)(e,1)(t,1)(',1)(s,1)( ,1)(c,1)(o,1)(u,1)(n,1)(t,1)( ,1)(s,1)(o,1)...`}
3. The function `reduceByKey` applies addition to the second component of those pairs whose first component is the same.
4. The action `collect` allows us to print the final result.

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

- We present the streaming version of the previous function.

```
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 = 3))
  val css : DStream[Char] = 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

- List of 4 characters arrive in each batch interval.
- For each of these batches, we apply the previous count.
  $u \equiv \boxed{\{"let'"\}}\ \boxed{\{"s\ co"\}}\ \boxed{\{"unt\ "\}}\ \boxed{\quad\ldots\quad}\ \boxed{\{"n"\}}$

```
-------------------------                  -------------------------
Time: 1449638784400 ms      ...     Time: 1449638785400 ms
-------------------------                  -------------------------
(e,1)                                      (n,1)
(t,1)
(l,1)
(',1)
```

In this talk

- We present `sscheck`, a test-case generator for Spark Streaming.
- We illustrate it with examples.
- We outline the underlying theoretical basis, although the details are in the paper.
- Related work also waits for the interested listener in the paper.

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 works as follows:
    - Several inputs are generated randomly.
    - The tool checks whether the outputs fulfill the formula.
- 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 for Core Spark

- Is it possible to use PBT with Core Spark?
- Is is just an adaptation of the existing framework (ScalaCheck).
- We generate random RDDs, possibly using the random generators for the values contained in the RDDs.
- And formulas (usually in FOL) to check the results after applying the functions under test.

Property-based testing for Spark Streaming

- Properties for streams are not straightforward.
- We have to consider temporal relations:
  - Events happen after/at the same time that other events.
  - Events take a specific time to happen.
- We need a logic and a test-case generator that handles time.

LTL_{ss}

- LTL_{ss} is a variant of propositional lineal temporal logic where formulas $\varphi \in LTL_{ss}$ are defined as:

$$\begin{array}{rl} \varphi ::= & \bot \mid \top \mid p \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \varphi \rightarrow \varphi \mid \\ & X\varphi \mid \Diamond_t\varphi \mid \Box_t\varphi \mid \varphi\ U_t\ \varphi \end{array}$$

- The operators in the logic are:

  Next indicates that the property holds in the next state.

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

  Always *for the next n batches*, which indicates that a property holds for the next n batches.

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

LTL_{ss}: Logic for finite words

- The logic for finite words proves judgements $u, i \vDash \varphi : v$ for $u \in \Sigma^*$, $i \in \mathbb{N}^+$, and $v \in \{\top, \bot, ?\}$.
- A formula is evaluated to ? when the word (stream) under test is too short for the formula.
- Given the word $u \equiv \boxed{\{b\}}\ \boxed{\{b\}}\ \boxed{\{a, b\}}\ \boxed{\{a\}}$.
- $u \vDash \Box_4\ (a \vee b) : \top$, since either a or b is found in the first four states.
- $u \vDash \Box_5\ (a \vee b) : ?$, since the property holds until the word is consumed, but the user required more steps.
- $u \vDash \Box_2(b \to \Diamond_2\ a) : \bot$, since in the first state we have b but we do not have a until the 3rd state.
- The generator defined by the formula $\Box_2(b \to \Diamond_2\ a)$ would randomly generate words such as $\boxed{\{b\}}\ \boxed{\{a, b\}}\ \boxed{\{a\}}$, $\boxed{\{a\}}\ \boxed{\{a\}}\ \boxed{\{a\}}$, or $\boxed{\{a\}}\ \boxed{\{b\}}\ \boxed{\{a\}}$, among others.

LTL_{ss}: Next form

- Thanks to timeouts, an interesting property of LTL_{ss} is that it is possible to compute beforehand the length of the test to avoid inconclusive results.
- They also allow to express formula in next form.
- We say that a formula $\psi \in LTL_{ss}$ is in *next form* iff. it is built by using the following grammar:

$$\psi ::= \quad \bot \mid \top \mid p \mid \neg\psi \mid \psi \vee \psi \mid \psi \wedge \psi \mid \psi \to \psi \mid \; X\, \psi$$

LTL_{ss}: Next form

- The next form allows us to evaluate formulas in a stepwise way.
- The basic idea for each step is to analyze atomic formulas and consume next operators.
- Hence, we can easily generate new letters in each step.
- It also provides an efficient evaluation algorithm when using a lazy implementation.

Letter simplification

Definition (Letter simplification)

Given a formula ψ in next form and a letter $s \in \Sigma$, the function $ls(\psi, s)$ *simplifies* ψ *with* s as follows:

- $ls(\top, s) = \top$.
- $ls(\bot, s) = \bot$.
- $ls(p, s) = p \in s$.
- $ls(\neg\psi, s) = \neg ls(\psi)$.
- $ls(\psi_1 \vee \psi_2, s) = ls(\psi_1) \vee ls(\psi_2)$.
- $ls(\psi_1 \wedge \psi_2, s) = ls(\psi_1) \wedge ls(\psi_2)$.
- $ls(\psi_1 \to \psi_2, s) = ls(\psi_1) \to ls(\psi_2)$.
- $ls(X\psi, s) = \psi$.

Applying propositional logic when definite values are found, it is possible to use this algorithm to obtain a value for the formula as soon as possible.

Example: Banned users

- We generate random DStreams of pairs `(userId, boolean)` where the boolean value is `false` if the user has performed a malicious action at that moment.
- The property specifies a transformation of that input DStream into an output stream containing the user ids of banned users, which have been malicious at some previous moment in time.
- For that we use:
  - A generator that generates good batches, where no malicious behavior has happened, until a bad batch for a particular malicious id occurs.
  - After that we generate either good or bad batches.
  - A property that states:
    - We always get good inputs, until we ban the malicious id.
    - Each time we find a malicious id, it is banned forever.

Example: Banned users

```
def checkExtractBannedUsersList(testSubject :
      DStream[(UserId, Boolean)] => DStream[UserId]) = {
    val batchSize = 20
    val (headTimeout, tailTimeout, nestedTimeout) = (10, 10, 5)
    val (badId, ids) = (15L, Gen.choose(1L, 50L))
    val goodBatch = BatchGen.ofN(batchSize, ids.map((_, true)))
    val badBatch = goodBatch + BatchGen.ofN(1, (badId, false))
    val gen = BatchGen.until(goodBatch, badBatch, headTimeout) ++
          BatchGen.always(Gen.oneOf(goodBatch, badBatch), tailTimeout)

    type U = (RDD[(UserId, Boolean)], RDD[UserId])
    val (inBatch, outBatch) = ((_ : U)._1, (_ : U)._2)

    ...
  }
```

Example: Banned users

```
def checkExtractBannedUsersList(testSubject :
       DStream[(UserId, Boolean)] => DStream[UserId]) = {
    ...

    val formula : Formula[U] = {
      val badInput : Formula[U] = at(inBatch)(_ should
            existsRecord(_ == (badId, false)))
      val allGoodInputs : Formula[U] = at(inBatch)(_ should
            foreachRecord(_._2 == true))
      val badIdBanned : Formula[U] = at(outBatch)(_ should
            existsRecord(_ == badId))

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

    forAllDStream(gen)(testSubject)(formula)
  }
```

Example: Banned users

- Given the dummy implementation:

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

- The tool returns the following information:

```
-------------------------------------------
Time: 1452577112500 ms - InputDStream1 (20 records)
-------------------------------------------
(6,true)
(3,true)
...
-------------------------------------------
Time: 1452577113000 ms - InputDStream1 (20 records)
-------------------------------------------
(5,true)
(29,true)
...
16/01/11 21:38:33 WARN DStreamTLProperty: finished test case 0
  with result False
```

Example: Twitter

```
def getHashtagsOk = {
  type U = (RDD[Status], RDD[String])
  val hashtagBatch = (_ : U)._2

  val numBatches = 5
  val possibleHashTags = List("#spark", "#scala", "#scalacheck")
  val tweets = BatchGen.ofNtoM(5, 10,
               TwitterGen.tweetWithHashtags(possibleHashTags))
  val gen = BatchGen.always(tweets, numBatches)

  val formula : Formula[U] = always {
    at(hashtagBatch){ hashtags =>
      hashtags.count > 0 and
      ( hashtags should
         foreachRecord(possibleHashTags.contains(_)) ) }
  } during numBatches

  forAllDStream(gen)(TweetOps.getHashtags)(formula)
}.set(minTestsOk = 10).verbose
```

Example: Twitter

```
Time: 1452668590000 ms - InputDStream1 (7 records)
-------------------------------------------
Lmawirg khX kzuea #spark gvy qub
Xgo HBvne #spark q xmhm ozcmzwm ctymzbnq fhaf
btisyv #scalacheck Fv b auRsnep s e dc Nes yorYuj wd zLeab
lxo ucvhno le ikaZ #scalacheck
...
-------------------------------------------
Time: 1452668590000 ms
-------------------------------------------
#spark
#spark
#scalacheck
#scalacheck
#scala
#scala
#scalacheck

16/01/12 23:03:13 WARN DStreamTLProperty: finished test case 0
 with result True
```

Conclusions

- We have explored the idea of extending property-based testing with temporal logic and its application to testing programs developed with a stream processing system.
- We have decided to work with a concrete system, Spark Streaming, in our prototype.
- In this way the tests are executed against the actual test subject and in a context closer to the production environment where programs will be executed.
- We think this could help with the adoption of the system by professional programmers.
- For this reason we have used Specs2, a mature tool for behavior driven development, for dealing with the difficulties of integrating our logic with Spark and ScalaCheck.

Future work

- There are many open lines of future work.
- Moving to FOL.
- We also consider developing versions for other languages with Spark API, in particular Python.
- It would also be interesting supporting other SPS, like Apache Flink.
- Finally, we intend to explore other formalisms for expressing temporal and cyclic behaviors.