Property-based testing for Spark Streaming

A. Riesco and J. Rodríguez-Hortalá

Facultad de Informática
Universidad Complutense de Madrid, Madrid, Spain

Apache: Big Data Europe 2016
November 14th, 2016 - Seville

Motivation

- Most of the time in the software developing cycle is devoted to testing and debugging.
- Batch systems provide different testing approaches.
- However, the tools for testing streaming systems are still immature.
- This is unfortunate, since more complex systems require closer attention.

Motivation

- However, it also makes sense, since probably the tools themselves need to be more complex.
- It is difficult to find the adequate tradeoff between complexity and power.
- A notable exception is Spark testing base [2].

Motivation

- We present here *sscheck*, a property-based testing framework for Spark Streaming.
- sscheck allows users to define generators and properties in LTL.
- The current version works for Spark 1.6.2.
- It is implemented in Scala.
- It allows users to define generators and properties in LTL.

Summary

Scala

- Scala is a general purpose programming language.
- It is an object-oriented language with full support for functional programming.
- These features includes higher-order types, lazy evaluation, and pattern matching.
- Scala code is compiled to Java bytecode, so it can be run in the Java Virtual Machine.

Spark

- Apache Spark is an open source cluster computing framework.
- 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

- Spark is implemented in Scala.
- 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.

Spark

- The set of predefined RDD transformations include 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.
- Spark powers a stack of libraries including SQL and DataFrames, MLlib for machine learning, GraphX, and Spark Streaming.

Spark

- The following example implements the ‘letter count’ example, a variant of the word count example:

```
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

- The following example implements the ‘letter count’ example, a variant of the word count example:

```
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

- The following example implements the 'letter count' example, a variant of the word count example:

```
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

- The following example implements the ‘letter count’ example, a variant of the word count example:

```
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.
- Actions are impure, so idempotent actions are recommended in order to ensure a deterministic behavior even in the presence of recomputations.

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

- A 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)
```

Property-based testing

- In *Property-based testing* [1] 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

Scalacheck

- Scalacheck [3] is a library written in Scala and used for automated property-based testing of Scala programs.
- It is also fully integrated in the test framework specs2.

```
import org.scalacheck.Properties
import org.scalacheck.Prop.forAll

object StringSpecification extends Properties("String") {

  property("startsWith") = forAll { (a: String, b: String) =>
    (a+b).startsWith(a)
  }

  property("concatenate") = forAll { (a: String, b: String) =>
    (a+b).length > a.length && (a+b).length > b.length
  }
}
```

Scalacheck

- We obtain the following results when executed:

```
$ sbt test
+ String.startsWith: OK, passed 100 tests.
! String.concat: Falsified after 0 passed tests.
> ARG_0: ""
> ARG_1: ""
```

- How is the magic done?

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.
- Scalacheck provides basic generators:

```
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]
```

Scalacheck - Generators

- It is also possible to pick values with a given frequency:

```
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 - Generators

- We can generate lists and sets as well:

```
val unifRand: Gen[Int] = Gen.choose(Int.MinValue,Int.MaxValue)
val gL1: Gen[List[Int]] = Gen.listOf(unifRand)
val gL2: Gen[List[String]] = Gen.listOfN(5,Gen.alphaStr)

val gA: Gen[Array[Int]] = Gen.containerOf[Array,Int](unifRand)
val gS: Gen[Set[Int]] = Gen.containerOfN[Set,Int](10,unifRand)

val gT: Gen[(Int,Int)] = Gen.zip(unifRand,unifRand)

val gM: Gen[Map[Int,String]] =
        Gen.mapOfN[Int,String](5,unifRand,alphaStr))
```

Scalacheck - Generators

- Finally, we can create our own generators:

```
case class User(name: String, age: Int)
val userGen: Gen[User] =
  for {
    name <- Gen.alphaStr
    age  <- Gen.choose(0,200)
  } yield User(name, age)

val gLU1: Gen[List[User]] = Gen.listOfN[User](100, userGen)
```

Property-based testing for Core Spark

- Is it possible to use random testing for Core Spark?
- Is is just an adaptation of the existing framework.
- We can generate random inputs from lists by using parallelize.
- And then state formulas using them.

Property-based testing for Spark Streaming

- However, properties for streaming systems are not straightforward.
- We have to consider temporal relations:
  - Events happen after/at the same time as other events.
  - Events take a specific time to happen.
- A property in first order logic would require to state all the possible combinations.
- We need a logic (and a tool) that handles time.

Linear Temporal Logic

- Linear Temporal Logic (LTL) is an extension of propositional logic that includes operator for indicating that:
  - A property *always* holds ($\Box\varphi$).
  - A property *eventually* holds ($\Diamond\varphi$).
  - A property holds *until* some other property holds ($\varphi_1 \cup \varphi_2$).
  - A property holds in the *next* instant ($\bigcirc\varphi$).

Linear Temporal Logic

- We still have problems.
- First, these properties are defined for infinite streams, but in practice we test finite ones.
- Moreover, it is not straightforward to relate different events within a fixed window.
- We require a temporal logic that explicitly introduces time.

LTL_{ss}

- The operators in the logic are:

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

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

  Until 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 indicates that the property holds in the next state.

LTL_{ss}

Example

Assume the set of atomic propositions $AP \equiv \{a, b, c\}$ and the word $u \equiv \boxed{\{b\}}\,\boxed{\{b\}}\,\boxed{\{a, b\}}\,\boxed{\{a\}}$. Then we have the following results:

- $u \vDash (\Diamond_4\ c) : \bot$, since c does not hold in the first four states.

LTL_{ss}

Example

Assume the set of atomic propositions $AP \equiv \{a, b, c\}$ and the word $u \equiv \boxed{\{b\}}\ \boxed{\{b\}}\ \boxed{\{a, b\}}\ \boxed{\{a\}}$. Then we have the following results:

- $u \models (\Diamond_4\ c) : \bot$, since c does not hold in the first four states.
- $u \models (\Diamond_5\ c) : ?$, since we have consumed the whole word, c did not hold in those states, and the timeout has not expired.

LTL_{ss}

Example

Assume the set of atomic propositions $AP \equiv \{a, b, c\}$ and the word $u \equiv \boxed{\{b\}}\boxed{\{b\}}\boxed{\{a, b\}}\boxed{\{a\}}$. Then we have the following results:

- $u \models (\Diamond_4\ c) : \bot$, since c does not hold in the first four states.
- $u \models (\Diamond_5\ c) : ?$, since we have consumed the whole word, c did not hold in those states, and the timeout has not expired.
- $u \models \Box_4\ (a \vee b) : \top$, since either a or b is found in the first four states.

LTL_{ss}

Example

Assume the set of atomic propositions $AP \equiv \{a, b, c\}$ and the word $u \equiv \boxed{\{b\}}\ \boxed{\{b\}}\ \boxed{\{a, b\}}\ \boxed{\{a\}}$. Then we have the following results:

- $u \vDash (\Diamond_4\ c) : \bot$, since c does not hold in the first four states.
- $u \vDash (\Diamond_5\ c) : ?$, since we have consumed the whole word, c did not hold in those states, and the timeout has not expired.
- $u \vDash \Box_4\ (a \vee b) : \top$, 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.

LTL_{ss}

Example

Assume the set of atomic propositions $AP \equiv \{a, b, c\}$ and the word $u \equiv \boxed{\{b\}}\,\boxed{\{b\}}\,\boxed{\{a, b\}}\,\boxed{\{a\}}$. Then we have the following results:

- $u \models (\Diamond_4\ c) : \bot$, since c does not hold in the first four states.
- $u \models (\Diamond_5\ c) : ?$, since we have consumed the whole word, c did not hold in those states, and the timeout has not expired.
- $u \models \Box_4\ (a \vee b) : \top$, since either a or b is found in the first four states.
- $u \models \Box_5\ (a \vee b) : ?$, since the property holds until the word is consumed, but the user required more steps.
- $u \models \Box_5\ c : \bot$, since the proposition does not hold in the first state.

LTL_{ss}

Example

Assume the set of atomic propositions $AP \equiv \{a, b, c\}$ and the word $u \equiv \boxed{\{b\}}\,\boxed{\{b\}}\,\boxed{\{a, b\}}\,\boxed{\{a\}}$. Then we have the following results:

- $u \vDash (b\ U_2\ a) : \bot$, since a holds in the third state, but the user wanted to check just the first two states.

LTL_{ss}

Example

Assume the set of atomic propositions $AP \equiv \{a, b, c\}$ and the word $u \equiv \boxed{\{b\}}\,\boxed{\{b\}}\,\boxed{\{a, b\}}\,\boxed{\{a\}}$. Then we have the following results:

- $u \vDash (b\ U_2\ a) : \bot$, since a holds in the third state, but the user wanted to check just the first two states.
- $u \vDash (b\ U_5\ a) : \top$, since a holds in the third state and, before that, b held in all the states.

LTL_{ss}

Example

Assume the set of atomic propositions $AP \equiv \{a, b, c\}$ and the word $u \equiv \boxed{\{b\}}\,\boxed{\{b\}}\,\boxed{\{a, b\}}\,\boxed{\{a\}}$. Then we have the following results:

- $u \models (b\ U_2\ a) : \bot$, since a holds in the third state, but the user wanted to check just the first two states.
- $u \models (b\ U_5\ a) : \top$, since a holds in the third state and, before that, b held in all the states.
- $u \models \Box_4(a \rightarrow X\ a) : ?$, since we do not know what happens in the fifth state, which is required to check the formula in the fourth state (because of next).

LTL_{ss}

Example

Assume the set of atomic propositions $AP \equiv \{a, b, c\}$ and the word $u \equiv \boxed{\{b\}}\,\boxed{\{b\}}\,\boxed{\{a, b\}}\,\boxed{\{a\}}$. Then we have the following results:

- $u \vDash (b\ U_2\ a) : \bot$, since a holds in the third state, but the user wanted to check just the first two states.
- $u \vDash (b\ U_5\ a) : \top$, since a holds in the third state and, before that, b held in all the states.
- $u \vDash \Box_4(a \rightarrow X\ a) : ?$, since we do not know what happens in the fifth state, which is required to check the formula in the fourth state (because of next).
- $u \vDash \Box_2(b \rightarrow \Diamond_2\ a) : \bot$, since in the first state we have b but we do not have a until the third state.

LTL_{ss}

Example

Assume the set of atomic propositions $AP \equiv \{a, b, c\}$ and the word $u \equiv \boxed{\{b\}}\ \boxed{\{b\}}\ \boxed{\{a, b\}}\ \boxed{\{a\}}$. Then we have the following results:

- $u \vDash (b\ U_2\ a) : \bot$, since a holds in the third state, but the user wanted to check just the first two states.
- $u \vDash (b\ U_5\ a) : \top$, since a holds in the third state and, before that, b held in all the states.
- $u \vDash \Box_4(a \to X\ a) : ?$, since we do not know what happens in the fifth state, which is required to check the formula in the fourth state (because of next).
- $u \vDash \Box_2(b \to \Diamond_2\ a) : \bot$, since in the first state we have b but we do not have a until the third state.
- $u \vDash b\ U_2\ X\ (a \wedge X\ a) :\ \top$, since $X(a \wedge X\ a)$ holds in the second state (that is, $a \wedge X\ a$ holds in the third state, which can also be understood as a holds in the third and fourth states).

sscheck

- sscheck implements this logic for defining generators and properties.
- Generators use Scalacheck generators for generating batches.
- We can combine them with LTL_{ss} operators to define time constraints.
- In the same way, properties are defined by using LTL_{ss} operators.
- Temporal formulas refer to *logic time*, that is, they count the number of batches.

Example - Twitter

- Assume we have implemented some functions that take tweets as arguments and we want to test them.
- We can check whether all the hashtags are correctly extracted.
- We can check whether the top hashtag is properly computed.
- We can use a reference implementation (a regular expression) for checking that the implementation works for batches.

Example - Twitter

- The tested functions are from the ampcamp 3: `http://ampcamp.berkeley.edu/3/exercises/realtime-processing-with-spark-streaming.html`.
- We first define the generators: `https://goo.gl/6oLJPm`.
- Then the properties: `https://goo.gl/RUd77U`.
- The results are available at: `https://goo.gl/LLpgYl`.

Beyond propositional logic

- The problem when using LTL_{ss} is that it is *stateless*.
- That is, we cannot bind values appearing in the batches.
- Hence, we are forced to state properties that only refer to the data in the current batch.
- This problem is solved by using first-order logic.
- Formulas in higher-order logic can be of the form $\forall(x, y)\varphi$, where x binds values in the current batch and y binds the current execution time.
- It is possible to use bound variables in both temporal operators and atomic propositions.

First-order modal logic

Example

Assume the set of atomic propositions $AP \equiv \{a, b, c\}$, the set of variables $\mathcal{V} \equiv \{x, y, z\}$, and the word $u \equiv \boxed{(\{b\}, 0)}\ \boxed{(\{b\}, 2)}\ \boxed{(\{a, b\}, 3)}\ \boxed{(\{a\}, 6)}$. Then we have the following results:

- $u \vDash \forall(\_, x).\Box_{x+1} b :\ \top$, since the first state contains a b.
- $u \vDash \forall(\_, x).\Box_3\ x < 3 :\ \top$, since it is always the case that $0 < 3$.
- $u \vDash \Box_3 \forall(\_, x).x < 3 :\ \bot$, since in the third state we have the time is 3.
- $u \vDash \Diamond_3(\forall(x, \_).X\ \forall(y, z).(x = y \wedge z < 3)) :\ \top$, since the proposition in the first step (b) is equal to the one in the second state, and the time in the second step (2) is smaller than 3.
- $u \vDash \forall(\_, x)\ .\ \Box_{x+1} b\ U_3\ a :\ \bot$, since a is not found until the third state but the first formula does not hold in the second state.

Example - Twitter revisited

- We can now define more powerful properties on our Twitter example.
- More powerful properties including quantifiers are available here: `https://goo.gl/BnM0V1`.
- The traces are available here: `https://goo.gl/3fXGwk`.

Implementation notes

- An interesting point of LTL_{ss} formulas is that they can be translated into an equivalent formula that only contains *next* as temporal operator.
- Formulas in *next form* distinguish between the current state and all the following ones, represented by means of nested next operators.
- Hence, generators in next form can generate one batch at a time.
- Similarly, formulas can process just one batch.
- This behavior is possible thanks to the timeouts in temporal operators.
- It can be efficiently implemented using the lazy features in Scala.

Implementation notes

Definition (Next transformation)

Given an alphabet Σ and a formula $\varphi \in LTL_{ss}$, the function $nt(\varphi)$ computes another formula $\varphi' \in LTL_{ss}$, such that φ' is in *next form* and $\forall u \in \Sigma^*. u \models \varphi \iff u \models \varphi'$.

$$
\begin{array}{lcl}
nt(\top) & = & \top \\
nt(\bot) & = & \bot \\
nt(t) & = & t \\
nt(ap) & = & ap \\
nt(t_1 = t_2) & = & t_1 = t_2 \\
nt(\varphi_1 \vee \varphi_2) & = & nt(\varphi_1) \vee nt(\varphi_2) \\
nt(\varphi_1 \wedge \varphi_2) & = & nt(\varphi_1) \wedge nt(\varphi_2) \\
nt(\varphi_1 \rightarrow \varphi_2) & = & nt(\varphi_1) \rightarrow nt(\varphi_2)
\end{array}
$$

for $p \in AP$ and $\varphi, \varphi_1, \varphi_2 \in LTL_{ss}$.

Implementation notes

Definition (Next transformation)

Given an alphabet Σ and a formula $\varphi \in LTL_{ss}$, the function $nt(\varphi)$ computes another formula $\varphi' \in LTL_{ss}$, such that φ' is in *next form* and $\forall u \in \Sigma^*. u \vDash \varphi \iff u \vDash \varphi'$.

$$
\begin{array}{lcll}
nt(X\ \varphi) & = & X\ nt(\varphi) & \\
nt(\Diamond_1 \varphi) & = & nt(\varphi) & \\
nt(\Diamond_t \varphi) & = & nt(\varphi) \vee X\ nt(\Diamond_{t-1} \varphi) & \text{if } t \geq 2 \\
nt(\Box_1 \varphi) & = & nt(\varphi) & \\
nt(\Box_t \varphi) & = & nt(\varphi) \wedge X\ nt(\Box_{t-1} \varphi) & \text{if } t \geq 2 \\
nt(\varphi_1\ U_1\ \varphi_2) & = & nt(\varphi_2) & \\
nt(\varphi_1\ U_t\ \varphi_2) & = & nt(\varphi_2) \vee & \\
 & & (nt(\varphi_1) \wedge X\ nt(\varphi_1\ U_{t-1}\ \varphi_2)) & \text{if } t \geq 2 \\
nt(\forall(b, r).\varphi) & = & \forall(b, r).nt(\varphi) &
\end{array}
$$

for $p \in AP$ and $\varphi, \varphi_1, \varphi_2 \in LTL_{ss}$.

Implementation notes

- Lazy evaluation is specially important when dealing with variables in temporal operators:
- For example, the following formula cannot be further reduced:
  $nt((\forall(\_, x)\ .\ \Box_{x+1} b)\ U_3\ a) =$
  $a \vee (\forall(\_, x)\ .\ nt(\Box_{x+1} b) \wedge X\ nt((\forall(\_, x)\ .\ \Box_{x+1} b)\ U_2\ a))$

Implementation notes

Definition (Letter simplification)

Given a formula ψ in next form and a letter $s \equiv (v_1, v_2) \in \Sigma \times \mathbb{N}$, the function $ls(\psi, s)$ *simplifies* ψ *with* s as follows:

- $ls(b, s) = b$ if $b \in \{\top, \bot\}$.
- $ls(p, s) = p \in s$.
- $ls(\psi_1 \vee \psi_2, s) = ls(\psi_1, s) \vee ls(\psi_2, s)$.
- $ls(\psi_1 \wedge \psi_2, s) = ls(\psi_1, s) \wedge ls(\psi_2, s)$.
- $ls(\psi_1 \rightarrow \psi_2, s) = ls(\psi_1, s) \rightarrow ls(\psi_2, s)$.
- $ls(X\ \psi, s) = \psi$.
- $ls(\forall(b, r).\psi, (v_1, v_2)) = \psi[b \mapsto v_1][r \mapsto v_2]$.

Implementation notes

Example

We present the evaluation process for the formula above using the word $u \equiv \boxed{(\{b\},0)}\,\boxed{(\{b\},2)}\,\boxed{(\{a,b\},3)}\,\boxed{(\{a\},6)}$.

- $ls(a \vee (\forall(\_,x)\ .\ nt(\Box_{x+1} b) \wedge X\ nt((\forall(\_,x)\ .\ \Box_{x+1} b)\ U_2\ a)), (\{b\},0)) =$ $a \vee (\forall(\_,x)\ .\ nt(\Box_{x+1} b) \wedge X\ nt((\forall(\_,x)\ .\ \Box_{x+1} b)\ U_1\ a)$
- $ls(a \vee (\forall(\_,x)\ .\ nt(\Box_{x+1} b) \wedge X\ nt((\forall(\_,x)\ .\ \Box_{x+1} b)\ U_1\ a)), (\{b\},2)) =$ $b \wedge X\ nt(\Box_1 b) \wedge a$
- $ls(b \wedge X\ nt(\Box_1 b) \wedge a, (\{a,b\},3)) = \top \wedge nt(\Box_1 b) \wedge \top \equiv b$
- $ls(b, (\{a\},6)) = \bot$

Implementation notes

Definition (Random word generation)

Given a formula ψ in next form without quantifiers, the function *gen* generates a finite word u such that $u \vDash \varphi$. If different equations can be applied for a given formula any of them can be randomly chosen:

$$
\begin{array}{lcl}
gen(\top) & = & \emptyset \\
gen(\bot) & = & err \\
gen(p) & = & \{p\} \\
gen(\varphi_1 \vee \varphi_2) & = & gen(\varphi_1) \\
gen(\varphi_1 \vee \varphi_2) & = & gen(\varphi_2) \\
gen(\varphi_1 \wedge \varphi_2) & = & gen(\varphi_1) \cup gen(\varphi_2) \\
gen(\varphi_1 \to \varphi_2) & = & gen(\varphi_2) \\
gen(X\ \varphi) & = & \emptyset + gen(\varphi)
\end{array}
$$

These words can be extended by pairing each letter with a number generated by random monotonically increasing function, hence generating valid inputs.

Conclusions

- We have implemented a property-based testing framework for Spark Streaming.
- It allows users to check temporal properties.
- Logic time is handled, so we consider a batch a time unit.
- Besides the usual limitations of LTL, we provide a first-order approach for more complex properties.

Ongoing Work

- Release sscheck for Spark 2.0.
- Implement shrinking.
- Study similar approaches, like Apache Flink.
- Real users are required!
- Collaborations are welcomed!

References

K. Claessen and J. Hughes.
QuickCheck: a lightweight tool for random testing of Haskell programs.
Acm sigplan notices, 46(4):53–64, 2011.

Holden Karau.
Spark-testing-base.
http://spark-packages.org/package/holdenk/spark-testing-base, 2015.

R. Nilsson.
ScalaCheck: The Definitive Guide.
IT Pro. Artima Incorporated, 2014.

Thanks!

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

ariesco@fdi.ucm.es

juan.rodriguez.hortala@gmail.com