Temporal Random Testing for Spark Streaming

Adrián Riesco(✉) and Juan Rodríguez-Hortalá

Universidad Complutense de Madrid, Madrid, Spain
{ariesco,juanrh}@fdi.ucm.es

Abstract. With the rise of Big Data technologies, distributed stream processing systems (SPS) have gained popularity in the last years. Among them, Spark Streaming stands out as a particularly attractive option with a growing adoption in the industry. In this work we explore the combination of temporal logic and property-based testing for testing Spark Streaming programs, by adding temporal logic operators to ScalaCheck generators and properties. This allows us to deal with the time component that complicates the testing of Spark Streaming programs and SPS in general. In particular we propose a discrete time linear temporal logic for finite words, that allows to associate a timeout to each temporal operator in order to increase the expressiveness of generators and properties. Finally, our prototype is presented with some examples.

Keywords: Stream processing systems · Spark streaming · Property-based testing · Random testing · Linear temporal logic · Scala · Big data

1 Introduction

With the rise of Big Data technologies [14], distributed stream processing systems (SPS) [1,14,25] have gained popularity in the last years. These systems are used to continuously process high volume streams of data, with applications ranging from anomaly detection [1], low latency social media data aggregation [14], or the emergent IoT market. Although the first precedents of stream processing systems come back as far as the early synchronous data-flow programming languages like Lutin [18] or Lustre [10], with the boom of SPS a plethora of new systems have arisen [12,20,25], characterized by a distributed architecture designed for horizontal scaling. Among them Spark Streaming [25] stands out as a particularly attractive option, with a growing adoption in the industry. In this work we focus on Spark Streaming. Spark [24] is a distributed processing engine that is quickly consolidating as an alternative to Hadoop MapReduce [14], due to an extended memory hierarchy that allows for an increased performance in

This research has been partially supported by MINECO Spanish projects *StrongSoft* (TIN2012-39391-C04-04), CAVI-ART (TIN2013-44742-C4-3-R), and *TRACES* (TIN2015-67522-C3-3-R), and by the Comunidad de Madrid project N-Greens Software-CM (S2013/ICE-2731).

E. Ábrahám and M. Huisman (Eds.): IFM 2016, LNCS 9681, pp. 393–408, 2016.
DOI: 10.1007/978-3-319-33693-0_25

```
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))
```

Fig. 1. Letter count in spark

many situations, and a collection-based higher level API inspired in functional programming that together with a "batteries included" philosophy accelerates the development of Big Data processing applications. These "batteries" include libraries for scalable machine learning, graph processing, a SQL engine, and Spark Streaming. The core of Spark is a batch computing framework [24], that is based on manipulating so called Resilient Distributed Datasets (RDDs), which provide a fault tolerant implementation of distributed multisets. Computations are defined as transformations on RDDs, that should be deterministic and side-effect free, as the fault tolerance mechanism of Spark is based on its ability to recompute any fragment (partition) of an RDD when needed. Hence Spark programmers are encouraged to define RDD transformations that are pure functions from RDD to RDD, and the set of predefined RDD transformations includes typical higher-order functions like map, filter, etc., as well as 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 actions are impure, but idempotent actions are recommended in order to ensure a deterministic behavior even in the presence of recomputations triggered by the fault tolerance or speculative task execution mechanisms.[1] Spark is written in Scala and offers APIs for Scala, Java, Python, and R; in this work we focus on the Scala API. The example in Fig. 1 uses the Scala Spark shell to implement a variant of the famous word count example that in this case computes the number of occurrences of each character in a sentence. For that we use `parallelize`, a feature of Spark that allows us to create an RDD from a local collection, which is useful for testing. We start with a set of chars distributed among 3 partitions, we pair each char with a 1 by using `map`, and then group by first component in the pair and sum by the second by using `reduceByKey` and the addition function (`_+_`), thus obtaining a set of (char, frequency) pairs. We collect this set into an `Array` in the driver with `collect`.

These notions of transformations and actions are extended in Spark Streaming from RDDs to DStreams (Discretized Streams), which 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 in the sense that 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. The code in Fig. 2 is the streaming version of the code in Fig. 1. Here we want to process a DStream of characters, where batches are obtained by

[1] See https://spark.apache.org/docs/latest/programming-guide.html for more details.

```
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)
```

Fig. 2. Letter count in spark streaming

splitting a String into pieces by making groups (RDDs) of 4 consecutive characters, using `grouped`. We use the testing utility class `QueueInputDStream`, which generates batches by picking RDDs from a queue, to generate the input DStream by parallelizing each substring into an RDD with 3 partitions. The program is executed using the local master mode of Spark, which replaces slave nodes in a distributed cluster by threads, which is useful for developing and testing.

The Problem of Testing. As the field has grown mature, several standard architectures for streaming processing like the Lambda Architecture [14] or reactive streams [12] have been proposed for implementing a cost effective, always up-to-date view of the data that allows the system to react on time to events. These architectures deal in different ways with trade-offs between latency, performance, and system complexity. The bar is also raised by the sophistication of the algorithms involved. To keep up with the speed on the input data stream, approximate algorithms with sublinear performance are used, even for otherwise simple aggregations [8]. Similarly, specialized machine learning and data stream mining algorithms are adapted to the stream processing context [15].

Moreover, dealing with time and events makes SPS-based programs intrinsically hard to test. There are several proposals in the literature that deal with the problem of testing and modeling systems that deal with time. In this work, we focus on Pnueli's approach [17] based on the use of temporal logic for testing reactive systems. Our final goal is facilitating the adoption of temporal logic as an every day tool for testing SPS-based programs. But, how could we present temporal logic in a way accessible to the average programmer? We propose exploring how property-based testing (PBT) [7], as realized in ScalaCheck [16], can be the answer, using it as a bridge between formal logic and software development practices like test-driven development (TDD) [5]. The point is that PBT is a testing technique with a growing adoption in the industry, that already exposes first order logic to the programmer. In PBT a test is expressed as a property, which is a formula in a restricted version of first order logic that relates program input

and output. The testing framework checks the property by evaluating it against a bunch of randomly generated inputs. If a counterexample for the property is found then the test fails, otherwise it passes. The following is a "hello world" ScalaCheck property that checks the commutativity of addition:[2]

```
class HelloPBT extends Specification with ScalaCheck {
  def is = s2"""Hello world PBT spec, where int addition is commutative $intAdditionCommutative"""
  def intAdditionCommutative =
    Prop.forAll("x" |: arbitrary[Int], "y" |: arbitrary[Int]) { (x, y) => x + y === y + x
  }.set(minTestsOk = 100) }
```

PBT is based on *generators* (the functions in charge of computing the inputs) and *assertions* (the formula to be checked), that together with a *quantifier* form a *property*. In the example above the universal quantifier `Prop.forAll` is used to define a property that checks the assertion `x + y === y + x` for 100 values for `x` and `y` randomly generated by two instances of the generator `arbitrary[Int]`. Each of those pairs of values generated for `x` and `y` is called a *test case*, and a test case that refutes the assertions of a property is called a *counterexample*. Here `arbitrary` is a higher order generator that is able to generate random values for predefined and custom types. Besides universal quantifiers, ScalaCheck supports existential quantifiers — although these are not much used in practice [16,22]—, and logical operators to compose properties. PBT is a sound procedure to check the validity of the formulas implied by the properties, because if a counterexample is found it gives a definitive proof that the property is false. However, it is not complete, as there is no guarantee that the whole space of test cases is explored exhaustively, so if no counterexample is found then we cannot conclude that the property holds for all possible test cases that could had been generated. PBT is a lightweight approach that does not attempt to perform sophisticated automatic deductions, but it provides a very fast test execution that is suitable for the TDD cycle, and empirical studies [7,21] have shown that in practice random PBT obtains good results, with a quality comparable to more sophisticated techniques. This goes in the line of assuming that in general testing of non trivial systems is often incomplete, as the effort of completely modeling all the possible behaviors of the system under test with test cases is not cost effective in most software development projects, except for critical systems.

We already have programmers using first order logic to write the properties for the test cases. So to realize our proposal, all that is left is extending ScalaCheck to be able to use temporal logic operators from some variant of propositional LTL [6]. We will give the details for our temporal logic in the next section; for the time being consider that we have temporal operators with bounded time such as *always* φ *in* t, which indicates that φ must hold for the next t instants, or φ *until* ψ *in* t, which indicates that φ currently holds and, before t instants of time elapse, ψ must hold. That way we would obtain a propositional LTL formula extended with an outer universal quantifier over the test cases produced by the generators. This temporal logic should use discrete time, as DStreams are discrete. Also, the logic should fit the simple property checking

[2] Here we use the integration of ScalaCheck with the Specs2 [21] testing library.

mechanism of PBT, that requires fast evaluation of each test case. For this reason we use a temporal logic for finite words, like those used in the field of runtime verification [13], instead of using infinite ω-words as usual in model checking. Although any Spark DStream is supposed to run indefinitely, so it might well be modeled by an infinite word, in our setting we only model a finite prefix of the DStream. This allows us to implement a simple and fast sound procedure for evaluating test cases, because if a prefix of a DStream refutes a property then the whole infinite DStream also refutes the property. On the other hand the procedure is not complete because only a prefix of the DStream is evaluated, but anyway PBT was not complete in the first place. Hence our *test cases* will be *finite prefixes* of DStreams, that correspond to *finite words* in this logic. In Sect. 2 there is a precise formulation of our logic LTL_{ss}, for now let's consider a concrete example in order to get a quick grasp of our proposal.

Example 1. 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 the example simple, we assume that the input records are pairs containing a `Long` user id, and a `Boolean` value indicating whether the user has been honest at that instant. The output stream should include the ids of all the users that have been malicious now or in a previous instant. So, the test subject that implement this has type `testSubject : DStream[(Long, Boolean)] => DStream[Long])`. Note that a trivial, stateless implementation of this behavior that just keeps the first element of the pair fails to achieve this goal, as it is not able to remember which users had been malicious in the past.

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

To define a property that captures this behavior, we start by defining a *generator* for (finite prefixes of) the input stream. As we want this input to change with time, we use a temporal logic formula to specify the generator. We start by defining the atomic non-temporal propositions, which are generators of micro batches with type `Gen[Batch[(Long, Boolean)]]`, where `Batch` is a class extending `Seq` that represents a micro batch. We can generate good batches, where all the users are honest, and bad batches, where a user has been malicious. We generate batches of `20` elements, and use `15L` as the id for the malicious id:

```
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))
```

So far generators are oblivious to the passage of time. But in order to exercise the test subject thoroughly, we want to ensure that a bad batch is indeed generated, and that several arbitrary batches are generated after it, so we can check that once a user is detected as malicious, it is also consider malicious in subsequent instants. And we want all this to happen within the confines of the generated finite DStream prefix. This is where *timeouts* come into play. In our

temporal logic we associate a timeout to each temporal operator, that constrains the time it takes for the operator to resolve. For example in a use of *until* with a timeout of t, the second formula most hold before t instants have passed. Translated to generators this means that in each generated DStream prefix a batch for the second generator is generated before t batches have passed, i.e. between the first and the t-th batch. This way we facilitate that the interesting events had enough time to happen during the limited fraction of time considered during the evaluation of the property.

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

The resulting generator `gen` has type `Gen[PDStream[(Long, Boolean)]]`, where `PDStream` is a class that represents sequences of micro batches corresponding to a DStream prefix. Here `headTimeout` limits the number of batches before the bad batch occurs, while `tailTimeout` limits the number of arbitrary batches generated after that. The output stream is simply the result of applying the test subject to the input stream. Now we define the *assertion* that completes the property, as a *temporal logic formula*.

```
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)  }
```

Atomic non-temporal propositions correspond to assertions on the micro-batches for the input and output DStreams. That is expressed by the type alias `U` for the universe of atomic propositions. The functions `inBatch` and `outBatch` can be combined with `at` and a Specs2 assertion to define non-temporal atomic propositions like `allGoodInputs`, that states that all the records in the input DStream correspond to honest users. But we know that this will not be happening forever, because `gen` eventually creates a bad batch, so we combine the atomic propositions using temporal operators to state things like "we have good inputs and no id banned until we ban the bad id" and "each time we get a bad input we ban the bad id for some time." Here we use the same timeouts we used for the generators, to enforce the formula within the time interval where the interesting events are generated. Also, we use an additional `nestedTimeout` for the nested *always*. Timeouts for operators that apply an universal quantification on time, like *always*, limit the number of instants that the quantified formula needs to be true for the whole formula to hold. In this case we only have to check

`badIdBanned` for `nestedTimeout` batches for the nested `always` to be evaluated to true. Following ideas from the field of runtime verification [3,4], we consider a 3-valued logic where the third value corresponds to an inconclusive result used as the last resort when the input finite word is consumed before completely solving the temporal formula. Timeouts for universal time quantifiers help relaxing the formula so its evaluation is conclusive more often, while timeouts for existential time quantifiers like *until* make the formula more strict. We consider that it is important to facilitate expressing properties with a definite result, as quantifiers like *exists*, that often lead properties to an inconclusive evaluation, have been abandoned in practice by the PBT user community [16,22].

Finally, we use our temporal universal quantifier `forAllDStream` to put together the temporal generator and formula, getting a property that checks the formula for all the finite DStreams prefixes produced by the generator:

```
forAllDStream(gen)(testSubject)(formula).set(minTestsOk = 20)
```

The property fails as expected for the faulty trivial implementation above, and succeeds for a correct stateful implementation [19].

The rest of the paper is organized as follows: Sect. 2 describes our logic for testing stream processing systems, while Sect. 3 presents its implementation for Spark. Section 4 discusses some related work. Finally, Sect. 5 concludes and presents some subjects of future work. An extended version of this paper can be found in [19].

2 A Temporal Logic for Testing Spark Streaming Programs

We present in this section a linear temporal logic for defining properties on stream processing systems. We first define the basics of the logic and then show some interesting properties to prove formulas in an efficient way.

2.1 A Linear Temporal Logic with Timeouts for Practical Specification of Stream Processing Systems

We present in this section LTL_{ss}, a linear temporal logic that specializes LTL_3 [3] by allowing timeouts in temporal connectives. LTL_3 is an extension of LTL for runtime verification that takes into account that only *finite* executions can be checked, and hence a new value ? (inconclusive) can be returned if a property cannot be evaluated to either *true* ($\top$) or *false* ($\bot$). These values form a lattice with $\bot \leq ? \leq \top$.

LTL_{ss} pays closer attention than LTL_3 to finite executions by limiting the scope of temporal connectives. This allows users (i) to obtain either $\top$ or $\bot$ for any execution given it has a given length, which can be computed beforehand, and (ii) to define more precise formulas, since it is possible to indicate in an easy way the period when it is expected to hold. Moreover, as we will see in Sect. 2.2, we have devised an efficient algorithm for evaluating these formulas.

Formulae Syntax. In line with [3], assume a finite set of atomic propositions AP. We consider the alphabet $\Sigma = \mathcal{P}(AP)$. A finite word over Σ is any $u \in \Sigma^*$, i.e. any finite sequence of sets of atomic propositions. We use the notation $u = a_1 \dots a_n$ to denote that u has length n and a_i is the letter at position or time i in u. Each letter a_i corresponds to a set of propositions from AP that hold at time i. LTL_{ss} is a variant of propositional lineal temporal logic where formulas $\varphi \in LTL_{ss}$ are defined as:

$$\varphi ::= \bot \mid \top \mid p \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \varphi \rightarrow \varphi \mid X\varphi \mid \Diamond_t \varphi \mid \Box_t \varphi \mid \varphi \; U_t \; \varphi \mid \varphi \; R_t \; \varphi$$

for $p \in AP$, and $t \in \mathbb{N}^+$ a timeout. We will use the notation $X^n\varphi, n \in \mathbb{N}^+$, as a shortcut for n applications of the operator X to φ. The intuition underlying these formulas, that are formally defined below, is:[3]

- $X\varphi$, read "next φ," indicates that φ holds in the next state.
- $\Diamond_t \varphi$, read "eventually φ in t," indicates that φ holds in any of the next t states (including the current one).
- $\Box_t \varphi$, read "always φ in t," indicates that φ holds in all of the next t states (including the current one).
- $\varphi_1 \; U_t \; \varphi_2$, read "$\varphi_1$ holds until φ_2 in t," indicates that φ_1 holds until φ_2 holds in the next t states, including the current one. It is enough for φ_1 to hold until the state previous to the one where φ_2 holds.

Note that if $t = \infty$ then LTL_{ss} would correspond to LTL_3. However, since our programs can only process finite words, we only work with $t \in \mathbb{N}^+$. In this case it is possible to discard the inconclusive value and obtain only definite values if some constraints hold between the word and the formula being tested.

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 $v \in \{\top, \bot, ?\}$.

$$u, i \vDash \Diamond_t \varphi : \begin{cases} \top & \text{if } \exists k \in [i, min(i + (t-1), len(u))].\ u, k \vDash \varphi : \bot \\ \bot & \text{if } i + (t-1) \leq len(u) \wedge \forall k \in [i, i + (t-1)].\ u, k \vDash \varphi_1 : \bot \\ ? & \text{otherwise} \end{cases}$$

$$u, i \vDash \Box_t \varphi : \begin{cases} \top & \text{if } i + (t-1) \leq len(u) \wedge \forall k \in [i, i + (t-1)].\ u, k \vDash \varphi : \top \\ \bot & \text{if } \exists k \in [i, min(i + (t-1), len(u))].\ u, k \vDash \varphi : \bot \\ ? & \text{otherwise} \end{cases}$$

$$u, i \vDash \varphi_1 \; U_t \; \varphi_2 : \begin{cases} \top & \text{if } \exists k \in [i, min(i + (t-1), len(u))].\ u, k \vDash \varphi_2 : \top \wedge \\ & \qquad \forall j \in [i, k).\ u, j \vDash \varphi_1 : \top \\ \bot & \text{if } \exists k \in [i, min(i + (t-1), len(u))].\ u, k \vDash \varphi_1 : \bot \wedge \\ & \qquad \forall j \in [i, k].\ u, j \vDash \varphi_2 : \bot \\ \bot & \text{if } i + (t-1) \leq len(u) \wedge \forall k \in [i, i + (t-1)].\ u, k \vDash \varphi_1 : \top \wedge \\ & \qquad \forall l \in [i, min(i + (t-1), len(u))].\ u, l \vDash \varphi_2 : \bot \\ ? & \text{otherwise} \end{cases}$$

[3] Due to space limitations, the results for *release* are available in [19].

$$u, i \vDash X\varphi : \begin{cases} ? & \text{if } i = len(u) \\ v & \text{if } i < len(u) \wedge u, i+1 \vDash \varphi : v \end{cases}$$

The intuition underlying this definition is that, if the word is too short to check all the steps indicated by a temporal operator and neither $\top$ or $\bot$ can be obtained before finishing the word, then ? is obtained. Otherwise, the formula is evaluated to either $\top$ or $\bot$ just by checking the appropriate sub-word. In the following we say $u \vDash \varphi$ iff $u, 1 \vDash \varphi : \top$. Note that these formulas, when used as generators, produce finite words that fulfill the formula. In our tool these words are minimal in the sense that it stops as soon as the word fulfills the formula. By using temporal logic not only for the formulas but also for the data generators, we obtain a simple setting that is easy to grasp for average programmers.

Example 2. 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.
- $u \vDash \Box_5\ c : \bot$, since the proposition does not hold in the first state.
- $u \vDash (b\ U_2\ a) : \bot$, 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.
- $u \vDash \Box_4(a \rightarrow Xa) : ?$, 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.
- $u \vDash b\ U_2\ X(a \wedge Xa) : \top$, since $X(a \wedge Xa)$ holds in the second state (that is, $a \wedge Xa$ holds in the third state, which can also be understood as a holds in the third and fourth states).

Example 3. The generator defined by the formula $\Box_2(b \rightarrow \Diamond_2\ a)$ above 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.

We need now a decision procedure for evaluating formulas. Although we can use the formal definitions above to define it, we would obtain a procedure that requires the whole stream to be traversed before taking the next step, greatly worsening the performance of the tool. We propose in the next section a transformation that allows us to implement a stepwise algorithm. Details on the naïve procedure can be found in [19].

2.2 A Transformation for Stepwise Evaluation

In order to define this stepwise evaluation, it is worth noting that all the properties are finite (that is, all of them can be proved or disproved after a finite number of steps). It is hence possible to express any formula only using the temporal operator X, which leads us to the following definition.

Definition 1 (Next Form). *We say that a formula $\psi \in LTL_{ss}$ is in* next form *iff. it is built by using the following grammar:*

$$\psi ::= \bot \mid \top \mid p \mid \neg\psi \mid \psi \vee \psi \mid \psi \wedge \psi \mid \psi \rightarrow \psi \mid X\psi$$

It is possible to obtain the next form of any formula $\varphi \in LTL_{ss}$ as:

Definition 2 (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 $\forall u \in \Sigma^*.u \vDash \varphi \iff u \vDash \varphi'$.*

$$\begin{aligned}
nt(a) &= a, \quad a \in \{\top, \bot, p\} \\
nt(\neg\varphi) &= \neg nt(\varphi) \\
nt(\varphi_1 \ op \ \varphi_2) &= nt(\varphi_1) \ op \ nt(\varphi_2), \ \textit{with op either} \ \vee, \wedge, \ \textit{or} \ \rightarrow. \\
nt(X\varphi) &= X\,nt(\varphi) \\
nt(\Diamond_t\varphi) &= nt(\varphi) \vee X\,nt(\varphi) \vee \ldots \vee X^{t-1}nt(\varphi) \\
nt(\Box_t\varphi) &= nt(\varphi) \wedge X\,nt(\varphi) \wedge \ldots \wedge X^{t-1}nt(\varphi) \\
nt(\varphi_1 \ U_t \ \varphi_2) &= nt(\varphi_2) \vee (nt(\varphi_1) \wedge X\,nt(\varphi_2)) \vee \\
&\quad (nt(\varphi_1) \wedge X\,nt(\varphi_1) \wedge X^2 nt(\varphi_2)) \vee \ldots \vee \\
&\quad (nt(\varphi_1) \wedge X\,nt(\varphi_1) \wedge \ldots \wedge X^{t-2}nt(\varphi_1) \wedge X^{t-1}nt(\varphi_2))
\end{aligned}$$

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

It is easy to see that the formula obtained by this transformation is in *next form*, since it only introduces formulas using the X operator. The equivalence between formulas is stated in Theorem 1 (the proof is available in [19]):

Theorem 1. *Given an alphabet Σ and formulas $\varphi, \varphi' \in LTL_{ss}$, such that $\varphi' \equiv nt(\varphi)$, we have $\forall u \in \Sigma^*.u \vDash \varphi \iff u \vDash \varphi'$.*

Example 4. We show how to transform some of the formulas from Example 2:

- $nt(\Diamond_4\ c) = c \vee Xc \vee X^2c \vee X^3c$
- $nt(b\ U_2\ a) = a \vee (b \wedge Xa)$
- $nt(\Box_2(b \rightarrow \Diamond_2\ a)) = (b \rightarrow (a \vee Xa)) \wedge X(b \rightarrow (a \vee Xa))$
- $nt(b\ U_2\ X(a \wedge Xa)) = X(a \wedge Xa) \vee (b \wedge X^2(a \wedge Xa))$

Once the next form of a formula has been computed, it is possible to evaluate it for a given word just by traversing its letters. We just evaluate the atomic formulas in the present moment (that is, those properties that does not contain the next operator) and remove the next operator otherwise, so these properties will be evaluated for the next letter. This method is detailed as follows:

Definition 3 (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:*

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

Using this function and applying propositional logic when definite values are found it is possible to evaluate formulas in a step-by-step fashion.[4] This transformation gives also the intuition that inconclusive values can be avoided if we use a word as long as the number of next operators nested in the transformation plus 1. A formal definition for this property can be found in [19].

3 Temporal Logic for Property-Based Testing

Our prototype extends ScalaCheck to support LTL_{ss} formulas for testing Spark Streaming programs. We use Spark's local mode to execute the test locally, so it is limited by the computing power of a single machine, but can be easily integrated in a continuous integration pipeline (e.g. the one for this same project https://travis-ci.org/juanrh/sscheck). Besides, our system is able to test programs without any modification. The system is available at https://github.com/juanrh/sscheck/releases.

Mapping Spark Streaming Programs into LTL_{ss}. Instead of using wall-clock time, like e.g. in Specs2's future matchers [21], we consider the logical time as discretized by the batch interval. At each instant, for each DStream we can see an RDD for the current batch as it was computed instantaneously. In practice the synchronization performed by Spark Streaming makes it appear like that, when enough computing resources are available. We define our atomic propositions as assertions over those RDDs. We have implemented an algebraic data type as a Scala trait `Formula`, that is parameterized on a universe type for the alphabet. The universe is a tuple of RDDs with one component for each DStream: e.g. in the example at the end of Sect. 1 the universe was defined by the alias `type U = (RDD[Double], RDD[Long])`, where the first component is the current batch for the input DStream and the second the current batch for the output DStream. `Formula` has a child case class for each of the constructions in LTL_{ss}, with a couple of exceptions. $\bot$, $\top$, and atomic propositions are all represented by the case class `Now`, which is basically a wrapper for a function from the universe into a ScalaCheck `Prop.Status` value, that represents a truth value. We need a function because we have to repeatedly apply it to each of the batches that are generated for each DStream. We provide suitable Scala implicit conversions for defining these functions more easily, using specs2 matchers: for example, at the end of Sect. 1, the argument of the `always` used to define the value `formula` is implicitly converted into a `Now` object. The other exception is `Solved`, that

[4] Note that the value? is only reached when the word is consumed and this simplification cannot be applied.

is used to represented formulas that have been evaluated completely. Although LTL_{ss} is a propositional temporal logic, in our prototype we add an additional outer universal quantifier on the test cases, as usual in PBT, so the test passes iff none of the generated test cases is able to refute the formula. Currently we do not support nesting of first order ScalaCheck quantifiers inside LTL_{ss} formulas.

We have also implemented higher-order ScalaCheck generators corresponding to temporal operators, where each generated test case represents a finite prefix of a DStream. For that we use the classes `Batch[A]` and `PDStream[A]`—that stands for prefix DStream—extending `Seq[A]` and `Seq[Batch[A]]` with additional operations like batch-wise union of PDStream.

Evaluating Temporal Properties. We provide a trait `DStreamTLProperty` with a method `forAllDStream`, as described in Sect. 1, for specifying properties on functions that transform DStreams, using the logic LTL_{ss}. The class `Formula` has methods for computing the next form, and for performing a step in the letter simplification process from Definition 3 by consuming a value of the type of the universe. On property evaluation we use `TestInputStream` from [11] to transform each `PDStream[A]` into a `DStream[A]`, and apply the test subject to create a derived DStream. Then we register a `foreachRDD` action on the input DStream that updates a `Formula` object for each new generated batch. For each test case we create a fresh streaming context, which is important for test case isolation in stateful transformations. We then start the Spark streaming context to start the computation, and then run a standard ScalaCheck `forall` property to generate the test cases. As soon as a `Solved` formula with failing status is reached, we stop the streaming context and return a failing property, and so ScalaCheck reports the current test case as a counterexample for the formula.

The resulting system has a reasonable performance. On a more realistic example based on official Spark training (computing the most popular hastag in a stream of tweets[5]), our system evaluates 50 test cases in 2 min and 4 s running in an Intel i7-4810MQ CPU with 16 GB RAM. The `batchDuration` parameter can be tuned according to the power of the machine: smaller values for faster machines, to complete the test earlier, and bigger values for slower machines, so the machine has more time to compute each of the batches.

4 Related Work

We can consider the system presented in this paper an evolution of the data-flow approaches devised for reactive systems in the past decades; we focus here in Lustre [10] and Lutin [18], since we consider they present a number of features that are representative of this kind of systems. In fact, the idea underlying both stream processing systems and data-flow reactive systems is very similar:

[5] See https://github.com/juanrh/sscheck-examples/blob/master/src/test/scala/es/ucm/fdi/sscheck/spark/demo/twitter/TwitterAmpcampDemo.scala and https://github.com/juanrh/sscheck-examples/wiki/TwitterAmpcampDemo-execution for the execution logs.

precessing a potentially infinite input stream while generating an output stream. Moreover, they usually work with formulas considering both the current state and the previous ones, which are similar to the "forward" ones presented here. There are, however, some differences between these two approaches, being an important one that sscheck is executed in a parallel way using Spark.

Lustre is a programming language for reactive systems that is able to verify safety properties by generating random input streams. The random generation provided by sscheck is more refined, since it is possible to define some patterns in the stream in order to verify some behaviors that can be omitted by purely random generators. Moreover, Lustre specializes in the verification of critical systems and hence it has features for dealing with this kind of systems, but lacks other general features as complex data-structures, although new extensions are included in every new release. On the other hand, it is not possible to formally verify systems in sscheck; we focus in a lighter approach for day-to-day programs and, since it supports all Scala features, its expressive power is greater. Lutin is a specification language for reactive systems that combines constraints with temporal operators. Moreover, it is also possible to generate test cases that depend on the previous values that the system has generated. First, these constraints provide more expressive power than the atomic formulas presented here, and thus the properties stated in Lutin are more expressive than the ones in sscheck. Although more expressive formulas are an interesting subject of future work, we have focused in this work in providing a framework where the properties are "natural" even for engineers who are not trained in formal methods; once we have examined the success of this approach we will try to move into more complex properties. Second, our framework completely separates the input from the output, and hence it is not possible to share information between these streams. Although sharing this information is indeed very important for control systems, we consider that stream processing systems usually deal with external data and hence this relation is not so relevant for the present tool. Finally, note that an advantage of sscheck consists in using the same language for both programming and defining the properties.

In a similar note, we can consider runtime monitoring of synchronous systems like Lola [9], a specification language that allows the user to define properties in both past and future LTL. Lola guarantees bounded memory for monitoring and allows the user to collect statistics at runtime. On the other hand, and indicated above, sscheck allows to implement both the programs and the test in the same language and provides PBT, which simplifies the testing phase, although actual programs cannot be traced. TraceContract [2] is a Scala library that uses a shallow internal DSL for implementing a logic for trace analysis. That logic is a hybrid between state machines and temporal logic, that is able to express both past time and future time temporal logic formulas, and that allows a form of first order quantification over the events that constitute the traces. On the other hand TraceContract is not able to generate test cases, and it is not integrated with any standard testing library like Specs2.

Regarding testing tools for Spark, the most clear precedent is the unit test framework Spark Test Base [11], which also integrates ScalaCheck for Spark but only for Spark core. To the best of out knowledge, there is no previous library supporting property-based testing for Spark Streaming.

5 Conclusions and Ongoing Work

In this paper 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. Instead of developing an abstract model of stream processing systems that could be applied to any particular implementation and performing testing against a translation of actual programs into that model, 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, as it integrates more naturally with the tool set employed in disciplines like test driven development. For this same reason we have used Specs2, a mature tool for behavior driven development, for dealing with the difficulties integrating of our logic with Spark and ScalaCheck. Along the way we have devised the novel finite-word discrete-time linear temporal logic LTL_{ss}, in the line of other temporal logics used in runtime verification. We think it allows to easily write expressive and strict properties about temporal aspects of programs.

Our next movement will be showing the system to programmers and draw conclusions from their opinions and impressions. There are many open lines of future work. On the practical side our prototype still needs some work to get a robust system. Also, adding support for arbitrary nesting of ScalaCheck `forall` and `exists` quantifiers inside LTL_{ss} formula would be an interesting extension. We also consider developing versions for other languages with Spark API, in particular Python, or supporting other SPS, like Apache Flink. Besides, we plan to explore whether the execution of several test cases in parallel minimize the test suite execution time. In the theoretical side, we should give a formal characterization of the language generated by our generators. Finally, we intend to explore other formalisms for expressing temporal and cyclic behaviors [23].

References

1. Akidau, T., Balikov, A., Bekiroğlu, K., Chernyak, S., Haberman, J., Lax, R., McVeety, S., Mills, D., Nordstrom, P., Whittle, S.: MillWheel: fault-tolerant stream processing at internet scale. Proc. VLDB Endowment **6**(11), 1033–1044 (2013)
2. Barringer, H., Havelund, K.: TraceContract: A scala DSL for trace analysis. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 57–72. Springer, Heidelberg (2011)
3. Bauer, A., Leucker, M., Schallhart, C.: Monitoring of real-time properties. In: Arun-Kumar, S., Garg, N. (eds.) FSTTCS 2006. LNCS, vol. 4337, pp. 260–272. Springer, Heidelberg (2006)

4. Bauer, A., Leucker, M., Schallhart, C.: The good, the bad, and the ugly, but how ugly is ugly? In: Sokolsky, O., Taşıran, S. (eds.) RV 2007. LNCS, vol. 4839, pp. 126–138. Springer, Heidelberg (2007)
5. Beck, K.: Test Driven Development: By Example. Addison-Wesley Professional, Boston (2003)
6. Blackburn, P., van Benthem, J., Wolter, F. (eds.): Handbook of Modal Logic. Elsevier, Philadelphia (2006)
7. Claessen, K., Hughes, J.: QuickCheck: A lightweight tool for random testing of Haskell programs. ACM Sigplan Not. **46**(4), 53–64 (2011)
8. Cormode, G., Muthukrishnan, S.: An improved data stream summary: The count-min sketch and its applications. J. Algorithms **55**(1), 58–75 (2005)
9. D'Angelo, B., Sankaranarayanan, S., Sánchez, C., Robinson, W., Finkbeiner, B., Sipma, H.B., Mehrotra, S., Manna, Z.: LOLA: runtime monitoring of synchronous systems. In: Proceedings of the 12th International Symposium on Temporal Representation and Reasoning, TIME, pp. 166–174. IEEE Computer Society (2005)
10. Halbwachs, N.: Synchronous programming of reactive systems. Springer International Series in Engineering and Computer Science, vol. 215. Kluwer Academic Publishers, Dordrecht (1992)
11. Karau, H.: Spark-testing-base (2015). http://blog.cloudera.com/blog/2015/09/making-apache-spark-testing-easy-with-spark-testing-base/
12. Kuhn, R., Allen, J.: Reactive Design Patterns. Manning Publications, Greenwich (2014)
13. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Logic Algebraic Program. **78**(5), 293–303 (2009)
14. Marz, N., Warren, J.: Big Data: Principles and Best Practices of Scalable Realtime Data Systems. Manning Publications Co., Stamford (2015)
15. Morales, G.D.F., Bifet, A.: SAMOA: Scalable advanced massive online analysis. J. Mach. Learn. Res. **16**, 149–153 (2015)
16. Nilsson, R.: ScalaCheck: The Definitive Guide. IT Pro, Artima Incorporated, Upper Saddle River (2014)
17. Pnueli, A.: Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends. In: de Bakker, J.W., de Roever, W.-P., Rozenberg, G. (eds.) Current Trends in Concurrency. LNCS, vol. 224, pp. 510–584. Springer, Heidelberg (1986)
18. Raymond, P., Roux, Y., Jahier, E.: Lutin: a language for specifying and executing reactive scenarios. EURASIP J. Emb. Syst. **2008**, 1–11 (2008). Article ID: 753821
19. Riesco, A., Rodríguez-Hortalá, J.: A lightweight tool for random testing of stream processing systems (extended version). Technical Report SIC 02/15, Departamento de Sistemas Informáticos y Computación de la Universidad Complutense de Madrid, September 2015. http://maude.sip.ucm.es/~adrian/pubs.html
20. Schelter, S., Ewen, S., Tzoumas, K., Markl, V.: All roads lead to Rome: optimistic recovery for distributed iterative data processing. In: Proceedings of the 22nd ACM international conference on Conference on information & knowledge management, pp. 1919–1928. ACM (2013)
21. Shamshiri, S., Rojas, J.M., Fraser, G., McMinn P.: Random or genetic algorithm search for object-oriented test suite generation? In: Proceedings of the on Genetic and Evolutionary Computation Conference, pp. 1367–1374. ACM (2015)
22. Venners, B.: Re: Prop.exists and scalatest matchers (2015). https://groups.google.com/forum/#!msg/scalacheck/Ped7joQLhnY/gNH0SSWkKUgJ
23. Wolper, P.: Temporal logic can be more expressive. Inf. Control **56**(1/2), 72–99 (1983)

24. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, p. 2. USENIX Assoc (2012)
25. Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., Stoica, I.: Discretized streams: fault-tolerant streaming computation at scale. In: Proceedings of the 24th ACM Symposium on Operating Systems Principles, pp. 423–438. ACM (2013)