A Lightweight Tool for Random Testing of Stream Processing Systems (Extended Version)*

Adrián Riesco and Juan Rodríguez-Hortalá

Technical Report SIC 02/15

Departamento de Sistemas Informáticos y Computación,
Universidad Complutense de Madrid

September 2015

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

Abstract

With the rise of Big Data technologies, distributed stream processing systems have gained popularity in the last years. These are used to continuously process streams of data from various sources, like customer activity or sensors measurements, to obtain an always up to date view of the data that allows the system to react on time to events. Stream processing systems, like any other system dealing with time and events, are hard to test. But we can find several proposals in the literature for dealing with those problems. In particular we focus on Pnueli's approach 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 the development of stream processing programs. To do that, we consider property-based testing (PBT), a random testing technique that has gained popularity in the software development industry, as a bridge between formal logic and software development practices like test driven development. As PBT only handles finite test cases, we propose a novel discrete time linear temporal logic for finite words. In order to increase its expressiveness for formulating stricter properties, in this logic we associate a timeout to each temporal connective, that determines the maximum time at which the corresponding sub-formulas should be solved to true for the whole formula to hold. We implement our logic as a library extending ScalaCheck for testing Spark Streaming programs.
Keywords: Stream processing systems, Spark, Random testing, Property-based testing, LTL, Temporal Logic, Scala, Big data

Contents

1 Introduction

With the rise of Big Data technologies [36], distributed stream processing systems (SPS) [1, 21, 39] have gained popularity in the last years. These systems are used to continuously process high volume streams of data. One of the earliest examples of the new generation systems from the Internet era is Millwheel [1], which was designed and used by Google for tasks like anomaly detection and cluster-health monitoring. Twitter is also a prominent user of stream processing systems like Apache Storm [21] and its successor Twitter Heron [30]—which was developed in-house by Twitter—, and has used both to process the massive continuous flow of tweets in the Twitter Firehose, computing approximate statistics about tweets with latencies of seconds, that are then used in data products like Twitter's trending topics. Another example is Yahoo's S4 [24], which was used for live parameter tuning of its search advertising system using the user traffic. Stream processing has also applications to security, since companies have started publishing large amounts of real-time data on this subject to share information among them and improve their results (see e.g. the IBM page for this aim, `https://exchange.xforce.ibmcloud.com/`). Finally, the resurge of the IoT market promises new applications of stream processing to sensor data, or its application to machine-to-machine communication [11]. But the first precedents of stream processing systems come back as far as the early synchronous dataflow programming languages like Lucid [35] or Lustre [15]

As the field has grown mature, several standard architectures for streaming processing like the Lambda Architecture [20, 21], the Kappa Architecture [17], or reactive streams [18, 8] 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 tradeoffs between latency, performance, and system complexity. The bar is also raised by the complexity of the algorithms involved. Stream processing systems need to keep up with the speed on the input data stream, which often translates in strict sublinear performance requirements that can only be met by employing sophisticated and approximate algorithms, even for computing aggregations that otherwise would be simple in other settings, like offline batch computations [2, 9, 13]. Similary specialized machine learning and data stream mining algorithms had to be designed for its application to the stream processing context [23, 6].

When complex architectures and algorithms are involved, having good testing tools at our disposal becomes particularly important. Besides, stream processing systems, like any other system dealing with time and events, are intrinsically hard to test. We can find several proposals in the literature for dealing with those problems; in particular, in this work we consider Pnueli's approach [28, 29] 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 the average programmer when facing the task of employing a SPS to implement a transformation of data streams. For this reason we decided to use a technique that is familiar to programmers and choose property-based testing (PBT) [7], a random testing technique that has gained popularity in the software development industry. Classical unit testing with xUnit-like frameworks [22] is based on specifying input - expected output pairs; they compare the expected output with the one obtained by applying the corresponding input to the test subject. On the other hand, in property based testing we specify a property as a formula in a restricted version of first order logic, which relates input and outputs, and then the testing framework checks the property against a bunch of inputs that are randomly synthesized. If a counterexample disproving the property is found, then the test fails, otherwise it passes.

We use PBT as a bridge between formal logic and software development practices like test driven development [5], for which PBT is a valuable tool, accepted by the software development community. The key idea is extending PBT to allow using temporal logic operators to define properties. As PBT only handles finite test cases, we need a temporal logic for finite words, like those used in the field of runtime verification [19]. PBT implementations are lightweight systems that do not attempt to perform sophisticated automatic deductions, nor an exhaustive traversal of the search space like model checking systems. Instead, a PBT system just executes many random tests as quick as possible, to perform a fast scouting of the search space looking for a counterexample that might refute the property under test. Although this might look like a weak procedure, empirical studies [7, 32] have shown that in practice random PBT obtains good results, with a quality comparable to more sophisticated techniques. For this reason, each test case generated in our system represents a finite prefix of an infinite word, so a test execution corresponds to the evaluation of a temporal logic formula over that prefix.

There are several logics for finite words proposed in the field of runtime verification [3, 4]. Just like [3], 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. This led us naturally to considering temporal operators with timeout, that correspond to the number

of instants the corresponding sub-formulas should be solved for the whole formula to hold. That is the case for existential temporal quantifiers like those in an until or an eventually operator. For universal temporal quantifiers like those in an always operator the timeout corresponds to the number of instants the sub-formula should hold. We believe this kind of timeouts leads to natural properties and allows the programmer to express stricter properties that are easy to understand. We consider that it is very important to facilitate expressing strict properties with a definite result, as in the past quantifiers in PBT systems have been abandoned in practice by the development community, as it happened for example with the existential quantifier in ScalaCheck [25, 34]. We have formalized these ideas in a novel discrete time linear temporal logic for finite words. As we will see later on, an additional benefit of the use of timeouts is that it leads to a very simple evaluation mechanism. We have also developed higher order random data generators for temporal operators, because when using temporal formulas as properties, the need to specify inputs with a temporal behavior arises naturally. 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.

We have implemented our logic as a library extending ScalaCheck [25] for testing Spark Streaming programs [39] written in Scala [26]. We have chosen ScalaCheck because it is the de facto standard implementation of PBT for Scala. We also use specs2 [33], a general purpose testing library easily extensible and with a good integration with ScalaCheck. We have used Scala because Spark [38] is written in Scala, and new features and first released in its Scala API. Moreover, PBT is more accepted in the Scala community than in the Python or Java communities, for which Spark Streaming exposes an API. We have chosen Spark Streaming as the stream processing platform for our first prototype because, as we will see below, it is based on micro batches that define a discrete time, and so it is a natural fit to our discrete time temporal logic. Furthermore, Spark is rooted on functional programming and is native to Scala, for which PBT is quite popular, as we have just seen. Finally, Spark is a popular tool and its acceptance has increased quickly in the last years.

Let us conclude this section with a quick preview of our system. The function below defines a property that checks whether its argument `testSubject` is a function that transforms a stream of batches of doubles—i.e. a series of multisets of doubles—into a series of batches of a single element containing the number of elements in the input batch at that specific time. To do that we define a simple generator `gen` that produces random batches of 50 elements during 20 instants. The temporal formula defined in `formula` then states that for all the instants, the output batch should contain a single element, and that this only element should be equal to the number of elements in the input batch at the same instant.

```
def countForallAlwaysProp(
  testSubject : DStream[Double] => DStream[Long]) = {
    type U = (RDD[Double], RDD[Long])
    val (inBatch, transBatch) = ((_ : U)._1, (_ : U)._2)
    val numBatches = 20
    val formula : Formula[U] = always { (u : U) =>
      transBatch(u).count === 1 and
      inBatch(u).count === transBatch(u).first
    } during numBatches
    val gen = BatchGen.always(BatchGen.ofN(50, arbitrary[Double]),
                              numBatches)
    DStreamProp.forAll(gen)(testSubject)(formula)
  }.set(minTestsOk = 20).verbose
```

If we put this property in the context of the specs2 specification below, we can check that the default implementation provided by the `count` method of Spark Streaming's `DStream` class is correct, while the wrong implementation defined in the function `faultyCount`, which subtracts 1 from the number, fails to pass the test.

```
class StreamingFormulaDemo1
  extends Specification
  with SharedStreamingContextBeforeAfterEach
  with ResultMatchers
  with ScalaCheck {

  // Spark configuration
  override def sparkMaster : String = "local[5]"
  override def batchDuration = Duration(350)
```

	$\vee$	$\bot$	?	$\top$	$\wedge$	$\bot$	?	$\top$	$\neg$
$\bot$		$\bot$	?	$\top$		$\bot$	$\bot$	$\bot$	$\top$
?		?	?	$\top$		$\bot$	?	?	?
$\top$		$\top$	$\top$	$\top$		$\bot$	?	$\top$	$\bot$

Figure 1: Truth tables for the logical connectives in LTL_3

```
override def defaultParallelism = 4

def is = sequential ^ s2"""
  Simple demo Specs2 example for ScalaCheck properties with
  temporal formulas on Spark Streaming programs
    - where a simple property for DStream.count is
    a success ${countForallAlwaysProp(_.count)}
    - where a faulty implementation of the DStream.count
    fails ${countForallAlwaysProp(faultyCount) must beFailing} """

def faultyCount(ds : DStream[Double]) : DStream[Long] =
  ds.count.transform(_.map(_ - 1))
```

The rest of the paper is organized as follows: Section 2 describes our logic for testing Big Data systems, while Section 3 presents its implementation for Spark. Finally, Section 4 concludes and presents some subjects of future work.

2 A Temporal Logic for Testing Spark Streaming programs

We present in this section our 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 effectively evaluated to either *true* ($\top$) or *false* ($\bot$) in the given execution. These values form a lattice with $\bot \leq ? \leq \top$; we remind how the logical connectives work in this case in Figure 1.

LTL_{ss} pays closer attention 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 length that 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 Section 2.2, an efficient algorithm for evaluating these formulas can be implemented by taking advantage of its specific structure.

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. Similarly, an infinite word over Σ is any $w = a_1 a_2 \dots \in \Sigma^w$, i.e. an infinite sequence of sets of atomic propositions. LTL_{ss} is a variant of propositional lineal temporal logic and formulas $\varphi \in LTL_{ss}$ are defined as follows:

$$\begin{aligned} \varphi ::= \quad & \bot \mid \top \mid p \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \varphi \rightarrow \varphi \mid \\ & X\varphi \mid \Diamond_t \varphi \mid \Box_t \varphi \mid \varphi \; U_t \; \varphi \mid \varphi \; R_t \; \varphi \end{aligned}$$

for $p \in AP$, and $t \in \mathbb{N}^+ \cup \{\infty\}$ 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:

- $X\varphi$ 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, and φ_2 must hold. Note that it is enough for φ_1 to hold until the state previous to the one where φ_2 holds.
- $\varphi_1\ R_t\ \varphi_2$, read "$\varphi_2$ is released by φ_1 in t," indicates that φ_2 holds until both φ_1 and φ_2 hold in the next t states, including the current one. However, if φ_1 never holds and φ_2 always holds the formula holds as well.

Note that if $t = \infty$ then LTL_{ss} corresponds to LTL_3. According to Emerson's classification [12] LTL_{ss} is a propositional, linear, pointwise, discrete, and future tense temporal logic. However, since we can only generate finite words, we focus on $t \in \mathbb{N}^+$. As we show later, 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 $v \in \{\top, \bot, ?\}$.

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

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

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

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

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

We say $u \vDash \varphi$ iff $u, 1 \vDash \varphi : \top$. 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.

Example 1 *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 word,* c *did not hold in those states but 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.*
- $u \vDash (a\ R_2\ b) : \top$, *since b holds in all the required states.*
- $u \vDash (a\ R_4\ b) : \top$, *since a and b hold in the third state.*
- $u \vDash \Box_3(a \to Xa) : \top$, *since the formula holds in the first in the first three states (note that the fourth state is required, since the formula involves the next operator).*
- $u \vDash \Box_4(a \to Xa) : ?$, *since we do not know what happens in the fifth state, which is required to check the formula in the fourth state.*
- $u \vDash \Box_2(b \to \Diamond_2\ a) : \bot$, *since in the first state we have b but we do not have a until the third state.*
- $u \vDash b\ U_2\ X(a \wedge Xa) : \top$, *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).*

Once the formal definition has been presented, we require a decision procedure for evaluating formulas in an algorithmic way. Next, we present an algorithm inferred from the logic presented above.

Decision procedure for the logic for finite words Just like ScalaCheck [25] and any other testing tool of the QuickCheck family [7, 27], this decision procedure does not try to be complete for proving the veritative value of formulae, but just to be complete for failures. Hence, we can define an abstract rewriting system for reductions $u \vDash \varphi \rightsquigarrow^* v$ for v in the same domain as above. Given a letter $a \in \Sigma$, a word $u \in \Sigma^*$, a timeout $t \in \mathbb{N}^+$, and formulas $\varphi, \varphi_1, \varphi_2 \in LTL_{ss}$, we have the following rules:[1]

1. Rules for $u \vDash X\ \varphi$:
   1) $au \vDash X\ \varphi \quad \rightsquigarrow \quad u \vDash \varphi$
   2) $u \vDash X\ \varphi \quad \rightsquigarrow \quad ?$ otherwise
2. Rules for $u \vDash \Diamond_t\ \varphi$:
   1) $\epsilon \vDash \Diamond_t\ \varphi \quad \rightsquigarrow \quad ?$
   2) $u \vDash \Diamond_0\ \varphi \quad \rightsquigarrow \quad \bot$
   3) $u \vDash \Diamond_t\ \varphi \quad \rightsquigarrow \quad \top$ if $u \vDash \varphi \rightsquigarrow^* \top$
   4) $au \vDash \Diamond_t\ \varphi \quad \rightsquigarrow \quad u \vDash \Diamond_{t-1}\ \varphi$ if $au \vDash \varphi \rightsquigarrow^* \bot$
   5) $u \vDash \Diamond_t\ \varphi \quad \rightsquigarrow \quad ?$ otherwise
3. Rules for $u \vDash \Box_t\ \varphi$:
   1) $\epsilon \vDash \Box_t\ \varphi \quad \rightsquigarrow \quad ?$
   2) $u \vDash \Box_0\ \varphi \quad \rightsquigarrow \quad \top$
   3) $u \vDash \Box_t\ \varphi \quad \rightsquigarrow \quad \bot$ if $u \vDash \varphi \rightsquigarrow^* \bot$
   4) $au \vDash \Box_t\ \varphi \quad \rightsquigarrow \quad u \vDash \Box_{t-1}\ \varphi$ if $au \vDash \varphi \rightsquigarrow^* \top$
   5) $u \vDash \Box_t\ \varphi \quad \rightsquigarrow \quad ?$ otherwise
4. Rules for $u \vDash \varphi_1\ U_t\ \varphi_2$:
   1) $\epsilon \vDash \varphi_1\ U_t\ \varphi_2 \quad \rightsquigarrow \quad ?$
   2) $u \vDash \varphi_1\ U_0\ \varphi_2 \quad \rightsquigarrow \quad \bot$
   3) $u \vDash \varphi_1\ U_t\ \varphi_2 \quad \rightsquigarrow \quad \top$ if $u \vDash \varphi_2 \rightsquigarrow^* \top$
   4) $u \vDash \varphi_1\ U_t\ \varphi_2 \quad \rightsquigarrow \quad \bot$ if $u \vDash \varphi_1 \rightsquigarrow^* \bot \wedge u \vDash \varphi_2 \rightsquigarrow^* \bot$
   5) $au \vDash \varphi_1\ U_t\ \varphi_2 \quad \rightsquigarrow \quad u \vDash \varphi_1\ U_{t-1}\ \varphi_2$ if $au \vDash \varphi_1 \rightsquigarrow^* \top \wedge au \vDash \varphi_2 \rightsquigarrow^* \bot$
   6) $u \vDash \varphi_1\ U_t\ \varphi_2 \quad \rightsquigarrow \quad ?$ otherwise

[1] Formulas built with propositional operators just evaluate the sub-formulas and apply the connectives as shown in Figure 1.

5. Rules for $u \vDash \varphi_1\ R_t\ \varphi_2$:
   1) $\epsilon \vDash \varphi_1\ R_t\ \varphi_2 \quad \rightsquigarrow \quad ?$
   2) $u \vDash \varphi_1\ R_0\ \varphi_2 \quad \rightsquigarrow \quad \top$
   3) $u \vDash \varphi_1\ R_t\ \varphi_2 \quad \rightsquigarrow \quad \top$ if $u \vDash \varphi_1 \rightsquigarrow^* \top \wedge u \vDash \varphi_2 \rightsquigarrow^* \top$
   4) $u \vDash \varphi_1\ R_t\ \varphi_2 \quad \rightsquigarrow \quad \bot$ if $u \vDash \varphi_2 \rightsquigarrow^* \bot$
   5) $au \vDash \varphi_1\ R_t\ \varphi_2 \quad \rightsquigarrow \quad u \vDash \varphi_1\ R_{t-1}\ \varphi_2$ if $au \vDash \varphi_1 \rightsquigarrow^* \bot \wedge au \vDash \varphi_2 \rightsquigarrow^* \top$
   6) $u \vDash \varphi_1\ R_t\ \varphi_2 \quad \rightsquigarrow \quad ?$ otherwise

for ϵ the empty word. These rules follow this schema: (i) an inconclusive value is returned when the empty word is found; (ii) the formula is appropriately evaluated when the timeout expires; (iii) it evaluates the subformulas to check whether a value can be obtained; it consumes the current letter and continue the evaluation; and (iv) inconclusive is returned if the subformulas are evaluated to inconclusive as well, and hence the previous rules cannot be applied. Hence, note that these rules have conditions that depend on the future. This happens in rules with a condition involving $\rightsquigarrow^*$ that inspects not only the first letter of the word, i.e., what it is happening now, but also the subsequent letters, as illustrated by the following examples:

Example 2 *We recall the word* $u \equiv \boxed{\{b\}}\ \boxed{\{b\}}\ \boxed{\{a,b\}}\ \boxed{\{a\}}$ *from Example 1 and evaluate the following formulas:*

- $\boxed{\{b\}}\ \boxed{\{b\}}\ \boxed{\{a,b\}}\ \boxed{\{a\}} \vDash \Box_2(b \rightarrow \Diamond_2\ a) \rightsquigarrow \bot$, *because* b *holds but* $\boxed{\{b\}}\ \boxed{\{b\}}\ \boxed{\{a,b\}}\ \boxed{\{a\}} \vDash \Diamond_2\ a \rightsquigarrow \boxed{\{b\}}\ \boxed{\{a,b\}}\ \boxed{\{a\}} \vDash \Diamond_1\ a \rightsquigarrow \boxed{\{a,b\}}\ \boxed{\{a\}} \vDash \Diamond_0\ a \rightsquigarrow \bot$.
- $\boxed{\{b\}}\ \boxed{\{b\}}\ \boxed{\{a,b\}}\ \boxed{\{a\}} \vDash b\ U_2\ X(a \wedge Xa)) \rightsquigarrow \boxed{\{b\}}\ \boxed{\{a,b\}}\ \boxed{\{a\}} \vDash b\ U_1\ X(a \wedge Xa))$, *which requires to check the second and third states to check that the second formula does not hold. The we have* $\boxed{\{b\}}\ \boxed{\{a,b\}}\ \boxed{\{a\}} \vDash b\ U_1\ X(a \wedge Xa)) \rightsquigarrow \top$ *after checking the third and fourth states.*

To use this procedure as the basis for our implementation we would have to keep a list of suspended alternatives, that are pending of conditions that will only solve in the future. This is because, although we had all the batches for a generated test case corresponding to an input stream, the batches for output streams generated by transforming the input will be only generated after waiting the corresponding number of instances. This leads to a complex and potentially expensive computation, since many pending possible alternatives have to be kept open. Instead of using this approach, it would be much more convenient to define a *stepwise* method with transition rules that only inspects the first letter of the input word.

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:*

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

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

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

$$
\begin{array}{lcl}
nt(\top) & = & \top \\
nt(\bot) & = & \bot \\
nt(p) & = & p \\
nt(\neg\varphi) & = & \neg nt(\varphi) \\
nt(\varphi_1 \vee \varphi_2) & = & nt(\varphi_1) \vee nt(\varphi_2) \\
nt(\varphi_1 \wedge \varphi_2) & = & nt(\varphi_1) \wedge nt(\varphi_2) \\
nt(\varphi_1 \rightarrow \varphi_2) & = & nt(\varphi_1) \rightarrow nt(\varphi_2) \\
nt(X\varphi) & = & X\,nt(\varphi) \\
nt(\Diamond_t\varphi) & = & nt(\varphi) \vee X\,nt(\varphi) \vee \ldots \vee X^{t-1}nt(\varphi) \\
nt(\Box_t\varphi) & = & nt(\varphi) \wedge X\,nt(\varphi) \wedge \ldots \wedge X^{t-1}nt(\varphi) \\
nt(\varphi_1\ U_t\ \varphi_2) & = & nt(\varphi_2) \vee (nt(\varphi_1) \wedge X\,nt(\varphi_2)) \vee \\
 & & (nt(\varphi_1) \wedge X\,nt(\varphi_1) \wedge X^2 nt(\varphi_2)) \vee \ldots \vee \\
 & & (nt(\varphi_1) \wedge X\,nt(\varphi_1) \wedge \ldots \wedge X^{t-2}nt(\varphi_1) \wedge X^{t-1}nt(\varphi_2)) \\
nt(\varphi_1\ R_t\ \varphi_2) & = & (nt(\varphi_2) \wedge X\,nt(\varphi_2) \wedge \ldots \wedge X^{t-1}nt(\varphi_2)) \vee \\
 & & (nt(\varphi_1) \wedge nt(\varphi_2)) \vee (nt(\varphi_2) \wedge X(nt(\varphi_1) \wedge nt(\varphi_2))) \vee \\
 & & (nt(\varphi_2) \wedge X\,nt(\varphi_2) \wedge X^2(nt(\varphi_1) \wedge nt(\varphi_2))) \vee \ldots \vee \\
 & & (nt(\varphi_2) \wedge X\,nt(\varphi_2) \wedge \ldots \wedge X^{t-2}nt(\varphi_2) \wedge X^{t-1}(nt(\varphi_1) \wedge nt(\varphi_2))
\end{array}
$$

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

It is straightforward 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:

Lemma 1 *Given $n \in \mathbb{N}^+$, an alphabet Σ and formulas $\varphi, \varphi' \in LTL_{ss}$, if $\forall u \in \Sigma^*.u \vDash \varphi \iff u \vDash \varphi'$ then $\forall u, \in \Sigma^*.u, n \vDash \varphi \iff u, n \vDash \varphi'$*

Proof. Since $u \equiv a_1 \ldots a_m$, $m \in \mathbb{N}$, we distinguish the cases $n > m$ and $n \leq m$:

$n > m$ It is easy to see for all possible formulas that only ? can be obtained, so the property trivially holds.

$n \leq m$ Then we have $u' \equiv a_n \ldots a_m$ and, since we know that $u' \vDash \varphi \iff u' \vDash \varphi'$, the property holds.

□

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

Proof. We apply induction on formulas.

Base case. It is straightforward to see that the result holds for the constants $\top$ and $\bot$ and for an atomic predicate p.

Induction hypothesis. Given the formulas $\varphi_1, \varphi_2, \varphi_1', \varphi_2' \in sstl$, such that $\varphi_1' \equiv nt(\varphi_1)$ and $\varphi_2' \equiv nt(\varphi_2)$, we have $\forall u \in \Sigma^*.u \vDash \varphi_i \iff u \vDash \varphi_i'$, $i \in \{1, 2\}$.

Inductive case. We distinguish the different formulas in LTL_{ss}:

- For the formulas $\bot, \top, p, \neg\varphi_1, \varphi_1 \vee \varphi_2, \varphi_1 \wedge \varphi_2$, and $\varphi_1 \rightarrow \varphi_2$, is trivial to see that the result holds, since we just work in propositional logic.
- Given the formula $X\varphi_1$, we have to prove that $\forall u \in \Sigma^*.u \vDash X\varphi_1 \iff u \vDash X\varphi_1'$. This expression can be transformed using the definition for the satisfaction for the next operator into $\forall u \in \Sigma^*.u, 2 \vDash \varphi_1 \iff u, 2 \vDash \varphi_1'$, which holds by hypothesis and Lemma 1.
- Given the formula $\Diamond_t\varphi_1$, $t \in \mathbb{N}^+$, we have to prove that $\forall u \in \Sigma^*.u \vDash \Diamond_t\varphi_1 \iff u \vDash \varphi_1' \vee X\varphi_1' \vee \ldots \vee X^{t-1}\varphi_1'$. We distinguish the possible values for $u \vDash \Diamond_t\varphi_1$:
  - $u \vDash \Diamond_t\varphi_1 : \top$. In this case the property holds because there exists i, $1 \leq i \leq t$ such that $u, i \vDash \varphi_1 : \top$. Hence, $u \vDash X^{i-1}\varphi_1'$ by hypothesis and the definition of the next operator (note that for $i = 1$ we just have $u \vDash \varphi'$).
  - $u \vDash \Diamond_t\varphi_1 : \bot$. In this case $\forall i, 1 \leq i \leq t, u, i \vDash \varphi_1 : \bot$, so we have $u \vDash X^{i-1}\varphi_1' : \bot$ for $1 \leq i \leq t$ and the transformation is also evaluated to $\bot$.

- $u \vDash \Diamond_t \varphi_1 : ?$. In this case we have u of length n, $n < t$, and $\forall i, 1 \leq i \leq n, u, i \vDash \varphi_1 : \bot$. Hence, we have $u \vDash X^{i-1}\varphi_1' : \bot$ for $1 \leq i \leq n$ and $u \vDash X^{j-1}\varphi_1' : ?$ for $n+1 \leq j \leq t$. Hence, we have $\bot \vee \ldots \vee \bot \vee ? \vee \ldots \vee ? = ?$ and the property holds.

- The analysis for $\Box_t \varphi_1$ is analogous to the one for $\Diamond_t \varphi_1$.

- Given the formula $\varphi_1 \ U_t \ \varphi_2$, $t \in \mathbb{N}^+$, we have to prove that $\forall u \in \Sigma^*. u \vDash \varphi_1 \ U_t \ \varphi_2 \iff u \vDash \varphi_2' \vee (\varphi_1' \wedge X\varphi_2') \vee \ldots \vee (\varphi_1' \wedge X\varphi_1' \wedge \ldots \wedge X^{t-2}\varphi_1' \wedge X^{t-1}\varphi_2')$. We distinguish the possible values for $u \vDash \varphi_1 \ U_t \ \varphi_2$:

  - $u \vDash \varphi_1 \ U_t \ \varphi_2 : \top$. In this case we have from the definition that $\exists i, 1 \leq i \leq t$ such that $u, i \vDash \varphi_2 : \top$ and $\forall j, 1 \leq j < i, u, j \vDash \varphi_1 : \top$. Hence, applying the induction hypothesis we have $u \vDash \varphi_1' \wedge X\varphi_1' \wedge \ldots X^{i-2}\varphi_1' \wedge X^{i-1}\varphi_2' : \top$, and hence the property holds.
  - $u \vDash \varphi_1 \ U_t \ \varphi_2 : \bot$.
    - Case a) $\forall i, 1 \leq i \leq t. u, i \vDash \varphi_2 : \bot$. In this case we have $\forall i, 1 \leq i \leq t, u, i \vDash X^{i-1}\varphi_2' : \bot$, and hence the complete formula is evaluated to $\bot$.
    - Case b) $\exists i, 1 \leq i \leq t, \forall j, 1 < j \leq i. u, j \vDash \varphi_1 : \top$, $u, j \vDash \varphi_2 : \bot$ $u, i \vDash \varphi_1 : \bot$, and $u, i \vDash \varphi_2 : \bot$. In this case we have $\forall k, 0 \leq k < i, u \vDash X^k\varphi_2' : \bot$ and $u \vDash X^{i-1}\varphi_1' : \bot$ by inductive hypothesis. Hence, all the conjunctions are evaluated to $\bot$ and the property holds.
  - $u \vDash \varphi_1 \ U_t \ \varphi_2 : ?$. In this case we have u of length n, $n < t$, $\forall i, 1 \leq i \leq n, u, i \vDash \varphi_2 : \bot$, and $u, i \vDash \varphi_1 : \top$. Hence, the first i conjunctions in the transformation are evaluated to $\bot$ by the induction hypothesis, while the rest are evaluated to $?$ by the definition of the next operator and the property holds.

- The analysis for $\varphi_1 \ R_t \ \varphi_2$ is analogous to the one for $\varphi_1 \ U_t \ \varphi_2$, taking into account that formula also holds if φ_2 always holds.

□

Example 3 *We present here how to transform some of the formulas from Example 1:*

- $nt(\Diamond_4 \ c) = c \vee Xc \vee X^2c \vee X^3c$
- $nt(\Box_4 \ (a \vee b)) = (a \vee b) \wedge X(a \vee b) \wedge X^2(a \vee b) \wedge X^3(a \vee b)$
- $nt(b \ U_2 \ a) = a \vee (b \wedge Xa)$
- $nt(a \ R_2 \ b) = (a \wedge b) \vee (a \wedge X(a \wedge b))$
- $nt(\Box_3(a \rightarrow Xa)) = (a \rightarrow Xa) \wedge X(a \rightarrow Xa) \wedge X^2(a \rightarrow Xa)$
- $nt(\Box_2(b \rightarrow \Diamond_2 \ a)) = (b \rightarrow (a \vee Xa)) \wedge X(b \rightarrow (a \vee Xa))$
- $nt(b \ U_2 \ X(a \wedge Xa)) = X(a \wedge Xa) \vee (b \wedge X^2(a \wedge Xa))$

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.[2] In this way, we can solve the formulas from the previous example as follows:

Example 4 *We present here the evaluation process for the formulas in Example 2.*

- $\Box_2(b \rightarrow \Diamond_2\ a) \equiv (b \rightarrow (a \vee Xa)) \wedge X(b \rightarrow (a \vee Xa))$ *(from Example 3).*
  - $ls((b \rightarrow (a \vee Xa)) \wedge X(b \rightarrow (a \vee Xa)), \{b\}) = (\top \rightarrow (\bot \vee a)) \wedge (b \rightarrow (a \vee Xa)) \equiv a \wedge (b \rightarrow (a \vee Xa))$.
  - $ls(a \wedge (b \rightarrow (a \vee Xa)), \{b\}) = \bot \wedge (\top \rightarrow (\bot \vee a)) \equiv \bot$.
- $b\ U_2\ X(a \wedge Xa)) \equiv X(a \wedge Xa) \vee (b \wedge X^2(a \wedge Xa))$ *(from Example 3).*
  - $ls(X(a \wedge Xa) \vee (b \wedge X^2(a \wedge Xa)), \{b\}) = (a \wedge Xa) \vee (\top \wedge X(a \wedge Xa)) \equiv (a \wedge Xa) \vee (X(a \wedge Xa))$
  - $ls((a \wedge Xa) \vee (X(a \wedge Xa)), \{b\}) = (\bot \wedge a) \vee (a \wedge Xa) \equiv a \wedge Xa$.
  - $ls(a \wedge Xa, \{a, b\}) = \top \wedge a \equiv a$.
  - $ls(a, \{a\}) = \top$.

The next 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.[3] We define this *safe word length* as follows:

Definition 4 (Safe word length) *Given a formula $\varphi \in LTL_{ss}$, its longest required check $swl(\varphi) \in \mathbb{N}$ is the maximum word length of a word u such that we have $u \vDash \varphi \in \{\top, \bot\}$. It is defined as follows:*

$$
\begin{array}{lcl}
swl(\top) & = & 1 \\
swl(\bot) & = & 1 \\
swl(p) & = & 1 \\
swl(\neg\varphi) & = & swl(\varphi) \\
swl(\varphi_1 \vee \varphi_2) & = & max(swl(\varphi_1), swl(\varphi_2)) \\
swl(\varphi_1 \wedge \varphi_2) & = & max(swl(\varphi_1), swl(\varphi_2)) \\
swl(\varphi_1 \rightarrow \varphi_2) & = & max(swl(\varphi_1), swl(\varphi_2)) \\
swl(X\varphi) & = & swl(\varphi) + 1 \\
swl(\Diamond_t\varphi) & = & swl(\varphi) + (t-1) \\
swl(\Box_t\varphi) & = & swl(\varphi) + (t-1) \\
swl(\varphi_1\ U_t\ \varphi_2) & = & max(swl(\varphi_1), swl(\varphi_2)) + (t-1) \\
swl(\varphi_1\ R_t\ \varphi_2) & = & max(swl(\varphi_1), swl(\varphi_2)) + (t-1)
\end{array}
$$

Example 5 *We present here the safe word length for some of the formulas in Example 1:*

- $swl(\Diamond_4\ c) = 4$.
- $swl(\Box_4\ (a \vee b)) = 4$.
- $swl(b\ U_2\ a) = 2$.
- $swl(a\ R_2\ b) = 2$.
- $swl(\Box_3(a \rightarrow Xa)) = 4$.
- $swl(\Box_2(b \rightarrow \Diamond_2\ a)) = 3$.
- $swl(b\ U_2\ X(a \wedge Xa)) = 4$.

[2] Note that the value ? is only reached when the word is consumed and this simplification cannot be applied.
[3] Note that it is possible to avoid an inconclusive value with shorter words, so this is a *sufficient* condition.

3 Temporal Logic for Property-based testing of Spark Streaming programs

In this section we present our prototype, which extends ScalaCheck to allow formulas in our temporal logic for testing Spark Streaming programs. The current implementation uses Spark's local mode to mock a cluster, thus executing all the tests locally, so it is limited by the computing power of a single machine. Nevertheless, our system is able to test programs without any modification, and local execution allows for a trivial integration of our system in a continuous integration pipeline: in fact we do just that with Travis CI to automate the execution of our internal tests `https://travis-ci.org/juanrh/sscheck`. The system is available for download at `https://github.com/juanrh/sscheck/releases/tag/codeFreezeFlops16`.

3.1 Spark and Spark Streaming

With the boom of stream processing systems, a plethora of new systems have arisen, with proposals like Samza [14], Flink [31], Akka Streams [18] and Spark Streaming [39]. As our main goal is facilitating the adoption of temporal logic as a tool for test driven development of stream processing programs, we are interested in developing a prototype that runs the tests against actual programs in a concrete SPS, running in an execution environment as similar as possible to the production environment. This implies we have to choose a concrete SPS. There are to main families of SPS: one-at-a-time and micro batch systems [21]. Their main difference is that systems in the former category process each input record individually, while those in the latter category group several records into small batches that are generated at a fixed rate, and then processed together. Apart from the implications on latency and performance, micro batch systems fit naturally with a discrete time temporal logic like ours, by identifying each batch with a time instant.

Among them Spark Streaming [39] stands out as particularly attractive option for the functional programmer, not only because it is developed in the functional language Scala, but also because the abstractions it defines are fundamentally functional. Spark Streaming is a library for stream processing that extends the Spark distributed batch computing framework [38], that is based on manipulating distributed collections called RDDs—that stands for Resilient Distributed Datasets—, that correspond to distributed lists, which are a fundamental data structure in functional programming. We can define transformations on RDDs, that must be deterministic and free of side effects, as the fault tolerance mechanism of Spark is based on its capability for recomputing any fragment (partition) of an RDD when it is needed. Hence Spark programmers are encouraged to define RDD transformations that are pure functions from RDD to RDD, and a set of predefined transformation on RDD is available that includes typical higher order functions from functional programming like map, filter, etc., as well as aggregations by key for RDDs of key-value pairs. We can also use Spark actions, that allow us to collect results into the master computation node (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. These notions of transformations and actions are extended in Spark Streaming from RDDs to DStreams, which stands for Discretized Streams, which are nothing but series of RDDs corresponding to micro batches. This batches are generated at a fixed rate according to the configured *batch interval*. Spark Streaming in 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 when the batch interval is met. Actions on DStreams are also periodic and are executed synchronously for each micro batch. Besides, Spark is a very active project that has grown very quickly, and that it starting to reach maturity, but it is still open to experimentation. For all these reasons we have developed our first prototype for Spark Streaming.

3.2 System Description

We briefly discuss in this section the main points of our tool.

3.2.1 Mapping Spark Streaming programs into LTL_{ss}.

Instead of using actual clock time, like it is done for example in the future matchers in specs2 [33], we consider the logical time defined by the batch interval. Each discrete time instant corresponds to the time from the start and the end of a batch interval, where for each DStream we can see the RDD at that instant, as it was computed instantaneously. In practice that computation is not instantaneous, but the synchronization performed by Spark Streaming makes it seem like that, at least when enough computing

resources are available, and task scheduling is not delayed too much. On top of that we define our atomic propositions as assertions over the RDDs that correspond to the current batch for each DStream at the present time. Technically we have defined an algebraic data type as a Scala trait `Formula`, that is parameterized on a universe type for the alphabet. In Spark Streaming the universe will be a tuple of RDDs with one component for each DStream. We can see it in the example at the end of Sect. 1 where we consider the universe defined by the alias `type U = (RDD[Double], RDD[Long])`, as we are dealing with an input DStream of doubles and an output DStream of longs. Note that implies we could use `Formula` not only for Spark Streaming but also for any system that generates series of elements. `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 apply it several times, 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 is used to represented formulas that have been evaluated completely.

Even though 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. Note we treat ? like a success, thus understanding PBT as a sound but not complete refutation resolution procedure. This should be configurable in future versions of the system. Also, currently we do not support nesting of first order ScalaCheck quantifiers like `forall` or `exists` inside LTL_{ss} formulas.

3.2.2 Generating streams with temporal formulas.

As mentioned before, we have also implemented higher order random data generators corresponding to temporal operators, to cope with the need to specify inputs with a temporal behavior that arises naturally when writing temporal properties. Each test case corresponding to a prefix of a `DStream[A]` is represented as an objects of type `Seq[Seq[A]]`, where the outer sequence corresponds to the simulated DStream prefix, and the inner sequence to each of the batches. We use the custom classes `Batch[A]` and `PDStream[A]`—that stands for prefix DStream—extending `Seq[A]` and `Seq[Batch[A]]`, so we can add additional operations like batch-wise union of PDStream, which are useful for defining generators. These sequences are later parallelized using the Spark context in a custom `InputDStream`, that is basically a modification of the standard `QueueInputDStream`, that now allows changing dynamically the sequence of batches to be generated.

These generators produce finite words from a subset of the language generated by a LTL_{ss} formula φ, defined as the set of finite words u such that $u, 1 \vDash \varphi : \top$. It is only a subset, because the generated words are minimal in the sense that they contain the minimum letters to satisfy φ: for example for $\varphi_1 \; U_t \; \varphi_2$ no additional letter will be generated after a letter that satisfies φ_2. We have not yet developed a formal characterization of that property, but we think this goes in the line of the notion of good prefixes from [19].

3.2.3 Evaluating temporal properties.

Our system provides a function `DStreamProp.forAll` that can be used for specifying properties of functions that transforms DStreams using the logic LTL_{ss}. As seen in the example in Sect. 1 this function takes a generator of type `Gen[Seq[Seq[E1]]]`, a test subject of type `(DStream[E1]) => DStream[E2]`, and a LTL_{ss} formula as a `Formula[(RDD[E1], RDD[E2])]` object, and returns a ScalaCheck property that is successful iff all the generated test cases fulfill the formula. For evaluating formulas, `Formula` has a method `nextFormula` that returns its next form as an object of type `NextFormula`, which is a subtype of `Formula` that provides a method `consume` that takes a value of the type of the universe and performs a step in the letter simplification process from Def. 3, implementing the truth tables from Figure 1. When the property is executed, we use our custom `InputDStream` to create an input DStream, and apply the test subject to create a derived DStream. We register a `foreachRDD` action on the input DStream that updates a `Formula` object for each new generated batch. We then start Spark streaming context to start the computation, and then run a standard ScalaCheck `forall` property to generate the test cases. For each test case we start from a fresh copy of the next form of the input formula, and set our custom `InputDStream` to use the test case. 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, and tests can be executed faster in more powerful machines by setting more cores or a shorter batch interval by overriding `sparkMaster` and `batchDuration`. When the machine is too slow and then Spark Streaming starts throwing exceptions due to task not being able to be completed on time. This can be checked in the Spark GUI, and the `batchDuration` can be adjusted to a bigger value so slower machines have more time to compute the batches. As usual in PBT, in our system we focus on functional testing, setting aside performance aspects of the test subject.

3.3 Case study

We simulate a simple malicious users detection system to illustrate the power of our tool. We have, for each batch, the set of active users as input and the set of banned users as output. We create the class `StreamingFormulaDemo` using `ScalaCheck` to implement this system and check it. We define user identifiers are just `Long` values:

```
class StreamingFormulaDemo with ScalaCheck {
  type UserId = Long
```

We consider a stream of pairs with the user identifier and a Boolean value indicating whether they have a correct behavior. The system must include malicious users (those with `false` in the second element of the pair) into the banned set when detected and keep them for the whole computation, so they cannot use any service later on. However, a dummy implementation of this behavior that just keeps the first element of the pair fails to achieve this goal:

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

We implement the property `checkExtractBannedUsersList` to check it as follows. We first decide to fix the batch size to `20` and the id `15L` as the identifier of a malicious user; the rest of identifiers are generated values in the range `1L` to `50L`:

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

Then, we define some constants for the generators and the LTL_{ss} formula. We want to generate good inputs for some time `headTimeout`, then generate the bad input (so we check we can detect it), and then generate some more inputs for a time `tailTimeout` (in order to check that the id is kept as malicious). Using these values we create a generator that introduces good inputs *until* the bad one is found, and then some more inputs are included by using *always*:

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

We indicate now the type `U` for the data in the formula and distinguish between the input and output batches. Given the properties `allGoodInputs`, which indicates that all the inputs thus far are good, `badInput`, which indicates that the malicious user has been found in the input, and `badIdBanned`, which indicates that the malicious id is banned, we want to prove that good inputs are introduced until the bad one is found (that is, `allGoodInputs` U_{t_1} `badInput`) *and*, once it is found, it is always banned (that is, $\Box_{t_2}$(`badInput` $\rightarrow \Box_{t_3}$`badIdBanned`)). This is the formula written below, using the constants used above for the generator and an extra constant indicating for how long must hold inner formula:

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

  val formula : Formula[U] = {
    val allGoodInputs : Formula[U] = at(inBatch)(_ should
```

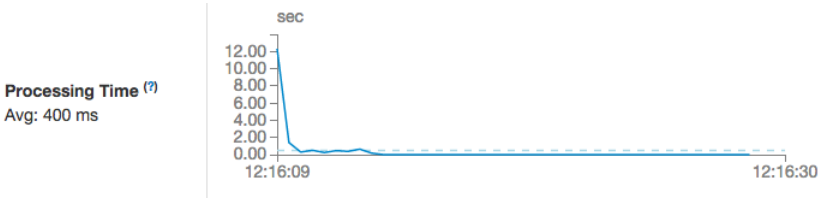

Figure 2: Performance for the malicious detection system

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

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

Finally, we put all together and check the property, which fails as expected.

```
   DStreamProp.forAll(gen)(testSubject)(formula)
 }.set(minTestsOk = 10).verbose
```

The tool shows a good performance, as shown in Figure 2, which is automatically computed by Spark.[4] Once the function is fixed it passes all the test, as can be seen in the repository at the beginning of the section.

4 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 a the novel finite word discrete time linear temporal logic LTL_{ss}, in the line of other temporal logics used in runtime verification, that we think it allows to easily write expressive and strict properties about temporal aspects of the programs. Regarding testing tools for Spark, the most clear precedent is [16], 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.

Our final goal is facilitating the adoption of temporal logic as an every day tool for the development of stream processing programs, and so our next movement will be showing the systems 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. We have not explored the use of our system with a cluster, but apart from the generators, which are created at the driver and then parallelized, there is nothing in particular in its design that would prevent using it with a cluster. In fact our custom specs2 matchers for RDDs are evaluated at the slave nodes, and the `sparkMaster` configuration setting could be employed for configuring the cluster execution. Finally, we plan to explore whether the execution

[4] On a computer with an Intel Core i5 2.5 GHz processor.

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 for temporal operators. Finally, we also intend to explore other formalisms for expressing temporal and cyclic behaviors [37, 10].

References

[1] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. MillWheel: fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044, 2013.

[2] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 20–29. ACM, 1996.

[3] A. Bauer, M. Leucker, and C. Schallhart. Monitoring of real-time properties. In *FSTTCS 2006: Foundations of Software Technology and Theoretical Computer Science*, pages 260–272. Springer, 2006.

[4] A. Bauer, M. Leucker, and C. Schallhart. The good, the bad, and the ugly, but how ugly is ugly? In *Runtime Verification*, pages 126–138. Springer, 2007.

[5] K. Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[6] A. Bifet, G. Holmes, R. Kirkby, and B. Pfahringer. Moa: Massive online analysis. *The Journal of Machine Learning Research*, 11:1601–1604, 2010.

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

[8] R. M. Community. Reactive streams, 2015. `http://www.reactive-streams.org/`.

[9] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[10] G. De Giacomo and M. Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In F. Rossi, editor, *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*. IJCAI/AAAI, 2013.

[11] T. Dunning and E. Friedman. *Practical Machine Learning: A New Look at Anomaly Detection*. O'Reilly Media, 2014.

[12] E. A. Emerson. Temporal and modal logic. *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, 995(1072):5, 1990.

[13] P. Flajolet, E. Fusy, O. Gandouet, and F. Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. *DMTCS Proceedings*, 0(1), 2008.

[14] M. Gorawski, A. Gorawska, and K. Pasterak. A survey of data stream processing tools. In *Information Sciences and Systems 2014*, pages 295–303. Springer, 2014.

[15] N. Halbwachs. *Synchronous programming of reactive systems*. Number 215. Springer Science & Business Media, 1992.

[16] Holden Karau. spark-testing-base. `http://spark-packages.org/package/holdenk/spark-testing-base`, 2015.

[17] J. Kreps. Questioning the lambda architecture. O'Reilly Radar, 2014. `http://radar.oreilly.com/2014/07/questioning-the-lambda-architecture.html`.

[18] R. Kuhn and J. Allen. Reactive design patterns, 2014.

[19] M. Leucker and C. Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293–303, 2009.

[20] N. Marz. How to beat the CAP theorem. Personal Blog, 2011. `http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html`.

[21] N. Marz and J. Warren. *Big Data: Principles and best practices of scalable realtime data systems*. Manning Publications Co., 2015.

[22] G. Meszaros. *xUnit test patterns: Refactoring test code*. Pearson Education, 2007.

[23] G. D. F. Morales and A. Bifet. SAMOA: Scalable advanced massive online analysis. *Journal of Machine Learning Research*, 16:149–153, 2015.

[24] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, pages 170–177. IEEE, 2010.

[25] R. Nilsson. *ScalaCheck: The Definitive Guide*. IT Pro. Artima Incorporated, 2014.

[26] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the scala programming language. Technical report, 2004.

[27] M. Papadakis and K. Sagonas. A PropEr integration of types and function specifications with property-based testing. In *Proceedings of the 10th ACM SIGPLAN workshop on Erlang*, pages 39–50. ACM, 2011.

[28] A. Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57. IEEE, 1977.

[29] A. Pnueli. *Applications of temporal logic to the specification and verification of reactive systems: a survey of current trends*. Springer, 1986.

[30] K. Ramasamy. Flying faster with twitter heron. The Official Twitter Blog, 2015. `https://blog.twitter.com/2015/flying-faster-with-twitter-heron`.

[31] S. Schelter, S. Ewen, K. Tzoumas, and V. Markl. 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*, pages 1919–1928. ACM, 2013.

[32] S. Shamshiri, J. M. Rojas, G. Fraser, and P. McMinn. Random or genetic algorithm search for object-oriented test suite generation? In *Proceedings of the 2015 on Genetic and Evolutionary Computation Conference*, pages 1367–1374. ACM, 2015.

[33] E. Torreborre. Specs2 user guide, 2014. `https://etorreborre.github.io/specs2/guide/SPECS2-3.6.2/org.specs2.guide.UserGuide.html`.

[34] B. Venners. Re: Prop.exists and scalatest matchers, 2015. `https://groups.google.com/forum/#!msg/scalacheck/Ped7joQLhnY/gNH0SSWkKUgJ`.

[35] W. W. Wadge and E. A. Ashcroft. *Lucid, the Dataflow Programming Langugage1*. Academic Press, 1985.

[36] T. White. *Hadoop: The definitive guide*. O'Reilly Media, 2012.

[37] P. Wolper. Temporal logic can be more expressive. *Information and Control*, 56(1/2):72–99, 1983.

[38] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. 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*, pages 2–2. USENIX Association, 2012.

[39] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, pages 423–438. ACM, 2013.