TLP **19** (4): 574–602, 2019. © Cambridge University Press 2019
doi:10.1017/S1471068419000012 First published online 19 February 2019

Property-Based Testing for Spark Streaming

A. RIESCO
Universidad Complutense de Madrid, Madrid 28040, Spain
(*e-mail:* ariesco@fdi.ucm.es)

J. RODRÍGUEZ-HORTALÁ
(*e-mail:* juan.rodriguez.hortala@gmail.com)

submitted 20 April 2017; revised 01 January 2019; accepted 09 January 2019

Abstract

Stream processing has reached the mainstream in the last years, as a new generation of open-source distributed stream processing systems, designed for scaling horizontally on commodity hardware, has brought the capability for processing high-volume and high-velocity data streams to companies of all sizes. In this work, we propose a combination of temporal logic and property-based testing (PBT) for dealing with the challenges of testing programs that employ this programming model. We formalize our approach in a discrete time temporal logic for finite words, with some additions to improve the expressiveness of properties, which includes timeouts for temporal operators and a binding operator for letters. In particular, we focus on testing Spark Streaming programs written with the Spark API for the functional language Scala, using the PBT library ScalaCheck. For that we add temporal logic operators to a set of new ScalaCheck generators and properties, as part of our testing library sscheck.

KEYWORDS: property-based testing, linear temporal logic, first-order modal logic, Spark Streaming, Scala

1 Introduction

With the rise of Big Data technologies (Marz and Warren 2015), distributed stream processing systems (SPS) (Akidau et al. 2013; Marz and Warren 2015) have gained popularity in the last years. This later generation of SPS systems, characterized by a distributed architecture designed for horizontal scaling, was pioneered by Internet-related companies that had to find innovative solutions to scale their systems to cope with the fast growth of the Internet. These companies are able to continuously process high-volume streams of data by using systems like MillWheel (Akidau et al. 2013), Apache Storm (Marz and Warren 2015), Heron (Ramasamy 2015), S4 (Neumeyer et al. 2010), and Samza (Gorawski et al. 2014). However, the first precedents of SPS come back as far as the early synchronous data-flow programming languages like Lutin (Raymond et al. 2008) or Lustre (Halbwachs 1992). A plethora of new distributed SPS have arisen in the last years, with proposals like Apache Flink (Carbone et al. 2015a), Akka Streams (Kuhn and Allen 2014), and Spark Streaming (Zaharia et al. 2013). Among them Spark Streaming stands out as a particularly popular option in the industry. A basic indicator for that

https://doi.org/10.1017/S1471068419000012 Published online by Cambridge University Press

is the evolution of the search terms for different SPS on Google Trends, showing Spark Streaming as the most popular SPS from January 2016 onwards (GoogleTrends 2018). Spark (Zaharia et al. 2012) is a distributed processing engine designed for processing large collections of data (White 2012). The core abstraction of Spark is the notion of Resilient Distributed Dataset (RDD), which provides a fault tolerant implementation of distributed collections. Spark Streaming is an SPS built on top of Spark, and it is based on the notion of DStreams (Discretized Streams), which are series of RDDs corresponding to splitting an input data stream into fixed time windows called microbatches, that are generated at a fixed rate according to a configured *batch interval*. Spark Streaming is synchronous in the sense that the batches for all DStreams in the program are generated at the same time, as the batch interval is met. See Appendix B in the supplementary material for a quick introduction to Spark and Spark Streaming.

In this work, we present a framework to test temporal properties on SPS. Among others, we consider that *safety* (something bad never happens) and *liveness* (something good eventually happens) properties are especially relevant in this kind of systems and might be useful for programmers. Specifically, we explore the problem of developing a testing library for Spark Streaming. We focus on Spark Streaming because its popularity implies a bigger set of potential users for our system, and in particular on its Scala API. We are interested in logic-based approaches, that nevertheless can be applied by software developers without the assistance of formal verification experts, as part of a test-driven development (TDD) cycle (Beck 2003). Testing an SPS-based program is intrinsically hard, because it requires handling time and events. Among the different proposals in the literature that tackle the problem of testing and modeling systems that deal with time, we follow Pnueli's approach (Pnueli 1986) that uses temporal logic for testing reactive systems. We define the logic LTL_{ss}, a variant of first-order linear temporal logic (LTL) (Blackburn et al. 2006) that is suitable for expressing Spark Streaming computations, which we expose to programmers as the sscheck library (Riesco and Rodríguez-Hortalá 2017b). This library extends ScalaCheck (Nilsson 2014), a popular property-based testing (PBT) (Claessen and Hughes 2011) library for Scala. In ScalaCheck, a test is expressed as a property, which is a formula in a restricted version of first-order logic that relates program inputs and outputs. Each quantifier in the formula is bound to a generator function that randomly produces values for some data type. The testing framework checks the property by evaluating it against a specified number of inputs that are produced by the generators. That provides a sound procedure for checking the validity of the formulas implied by the properties, that is not complete but that it is fast and lightweight enough to integrate in a TDD cycle – see Appendix C in the supplementary material for an overview of PBT and ScalaCheck. sscheck extends ScalaCheck with temporal logic operators that can be used both in the generator functions for the input DStreams, and in the quantified formula that relates input and output DStreams, which simplifies expressing complex conditions on the sequence of batches for each DStream.

As DStreams are discrete, LTL_{ss} uses *discrete time*. Our logic also considers finite words, like those used in the field of runtime verification (Leucker and Schallhart 2009), instead of infinite ω-words as usual in model checking. That allows us to easily integrate with the simple property checking mechanism of ScalaCheck. Although Spark DStreams are supposed to run indefinitely, so we might had modeled each DStream by an infinite

https://doi.org/10.1017/S1471068419000012 Published online by Cambridge University Press

word, in our setting, we only model a finite prefix of the DStream. This allows us to implement a simple, fast, and sound procedure for evaluating test cases, based on evaluating the property on the generated finite prefix. 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, a *test case* will be a tuple of *finite prefixes* of DStreams, which corresponds to a *finite word* in this logic, and the aforementioned external quantifier ranges over the domain of finite words. We provide a precise formulation for our logic LTL_{ss} in Section 2.1, while details on how to implement properties are presented in Section 3.2, but for now let us 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 receives a stream of events describing user activity in some system. The program outputs a stream with the identifiers of banned users, which are users that the system has detected as abusing the system based on their activity. 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 those users that have been malicious now or in a previous instant. So, the test subject that implements it has type `testSubject: DStream[(Long, Boolean)] => DStream[Long])`.

To define a property that captures the expected behavior, we start by using sscheck to define 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 microbatches with type `Gen[Batch[(Long, Boolean)]]`, where `Batch` is a class extending `Seq` that represents a microbatch. 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 user:

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

where `BatchGen.ofN` is a function that generates a batch with the specified number of elements, which are generated by the generator function in its second argument. In our logic LTL_{ss}, that corresponds to some predicate symbols under an interpretation structure $\mathcal{A} = (A, I)$ where the domain A is the set of all Scala expressions. As we are verifying a Spark Streaming program, given a $\mathbb{U} = \mathbb{RDD} \times \mathbb{RDD}$ for $\mathbb{RDD}$ the set of all Spark RDDs, we use timed letters in $\mathbb{U}^t = \mathbb{U} \times \mathbb{N}$ that correspond to an RDD for the input and another for the output DStream, together with the time at which the letter is produced. The interpretation function I would be defined as follows, assuming the usual meaning for the cardinal ($\#$) and inclusion ($\in$) operators for $\mathbb{RDD}$ and identifying Scala number literals with the corresponding numbers by abuse of notation:

$$
\begin{aligned}
&\mathit{goodId} = \{x \in \mathbb{Z} \mid 1 \leq x \leq 50\} \quad \mathit{goodElem} = \{e \in \mathbb{Z} \times \{\top, \bot\} \mid e = (x, \top) \wedge x \in \mathit{goodId}\}\\
&I(\mathit{goodBatch}) = \{((rdd_i, rdd_o), t) \in \mathbb{U}^t \mid \#(rdd_i) = 20 \wedge \forall e \in rdd_i.\ e \in \mathit{goodElem}\}\\
&I(\mathit{badBatch}) = \{((rdd_i, rdd_o), t) \in \mathbb{U}^t \mid \#(rdd_i) = 20\\
&\qquad\qquad \wedge \forall e \in rdd_i.\ e \in \mathit{goodElem} \vee e = (15, \bot)\}.
\end{aligned}
$$

https://doi.org/10.1017/S1471068419000012 Published online by Cambridge University Press

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 considered malicious in subsequent instants. Moreover, 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 must hold before t instants have passed, while the first one must hold until that moment. Translated to generators this means that in each generated DStream prefix, a batch for the second generator is generated before t batches have passed, that is, between the first and the tth 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 microbatches 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. That generator corresponds to the LTL_{ss} formula below, which is defined employing versions with timeout of classical temporal logic operators like $\Box_t$ (always for t batches), or U_t (until for t batches), as well as a new "consume" operator λ_x^t that is basically a variant of the classical "next" operator that binds a variable to the current letter and time in the input word:

$$(\lambda^o_{rdds}.\ goodBatch(rdds)\ U_{10}\ \lambda^o_{rdds}.\ badBatch(rdds))\ U_{100}$$
$$(\Box_{10}.\ \lambda^o_{rdds}.\ goodBatch(rdds) \vee badBatch(rdds)).$$

The output DStream is the result of applying the test subject to the input stream. 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 = {
  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 microbatches for the input and output DStreams. We use a syntax where the function `at` below is used with a projection function like `inBatch` or `outBatch` to apply an assertion on part of the current letter – for example, the batch for the current input. The assertions `foreachRecord` and `existsRecord` are custom Specs2 assertions that allow users to check whether a predicate holds or not for all or for any of the records in an RDD, respectively. This

https://doi.org/10.1017/S1471068419000012 Published online by Cambridge University Press

way, we are able 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 `allGoodInputs` 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. That corresponds to the following LTL_{ss} formula φ, assuming the interpretation of the predicate symbols *allGoodInputs*, *badInput*, *empty*, and *badIdBanned* as specified below:

$$
\begin{array}{l}
I(allGoodInputs) = \{(rdd_i, rdd_o) \in \mathbb{U} \mid \forall e \in rdd_i.\ e = (x, \top)\} \\
I(badInput) = \{(rdd_i, rdd_o) \in \mathbb{U} \mid \exists e \in rdd_i.\ e = (15, \bot)\} \\
I(noIdBanned) = \{(rdd_i, rdd_o) \in \mathbb{U} \mid \nexists e \in rdd_o\} \\
I(badIdBanned) = \{(rdd_i, rdd_o) \in \mathbb{U} \mid 15 \in rdd_o\} \\
(\lambda^o_{rdds}.\ allGoodInputs(rdds) \wedge \lambda^o_{rdds}.\ noIdBanned(rdds))\ U_{10}\ \lambda^o_{rdds}.\ badIdBanned(rdds) \\
\wedge\ \Box_{10}.\ \lambda^o_{rdds}.\ badInput(rdds) \rightarrow (\Box_5.\ \lambda^o_{rdds}.\ badIdBanned(rdds)).
\end{array}
$$

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 a faulty stateless implementation that is not able to remember which users had been malicious in the past, and succeeds for a correct stateful implementation (see Riesco and Rodríguez-Hortalá (2018) for details).

We carried out these ideas on the library sscheck (Riesco and Rodríguez-Hortalá 2017b), previously presented in the tool paper (Riesco and Rodríguez-Hortalá 2016b), and in a leading engineering conference (Riesco and Rodríguez-Hortalá 2016a). Moreover, sscheck has been discussed by others (Karau 2015c) and it has also been referred in books and technical blogs remarkable in the field (Karau 2015a; Karau and Warren 2017), showing that it presents a good performance and that it stands as an alternative choice for state-of-the-art testing frameworks.

The present paper extends the published works above by

- Improving the logic by (i) redefining the semantics of formulas using a first-order structure for letters that are evaluated under a given interpretation, (ii) introducing a new operator that allows us to bind the content and the time in the current batch, (iii) redefining the previous results for the new logic, and (iv) defining a new recursive definition that allows us to simplify formulas in a lazy way.
- Formally proving the theoretical results arising from the new formulation.
- Formalizing the generation of words from formulas.
- Providing extensive examples of sscheck properties, including safety and liveness properties.

https://doi.org/10.1017/S1471068419000012 Published online by Cambridge University Press

The rest of the paper is organized as follows: Section 2 describes our logic for testing SPS, while Section 3 presents its implementation for Spark. Section 4 discusses some related work. Finally, Section 5 concludes and presents some subjects of future work. The code of the tool, examples, and much more information are available in `https://github.com/juanrh/sscheck`. An extended version of this paper can be found in Riesco and Rodríguez-Hortalá (2018).

2 A logic for testing stream systems

We present in this section our LTL for defining properties on Spark Streaming programs. We first define the basics of the logic, then we show a stepwise formula evaluation procedure that is the basis for our prototype, and finally we formalize the generation of test cases from formulas.

2.1 An LTL with timeouts for practical specification of SPS

The basis of our proposal is the LTL_{ss} logic, an LTL that combines and specializes both LTL_3 (Bauer et al. 2006) and First-order Modal Logic (Fitting and Mendelsohn 1998), borrowing some ideas from TraceContract (Barringer and Havelund 2011). LTL_3 is an extension of *LTL* (Alur and Henzinger 1994) for runtime verification that takes into account that only *finite* executions can be checked, and hence a new value ? (inconclusive) can be returned in case a property cannot be effectively evaluated to either *true* ($\top$) or *false* ($\bot$) in the given execution, because the word considered was too short. These values form a lattice with $\bot \leq ? \leq \top$. LTL_{ss} uses the same domain as LTL_3 for evaluating formulas, and the same truth tables for the basic non-temporal logical connectives —see Riesco and Rodríguez-Hortalá (2018) for details. LTL_{ss} is also influenced by First-order Modal Logic, an extension to first order of the standard propositional modal logic approach (Blackburn et al. 2006). Although the propositional approach in Riesco and Rodríguez-Hortalá (2016b) was enough for generating new values and dealing with some interesting properties – including safety properties – we noticed that some other properties involving variables bound in previous letters – for example, some liveness properties – could not be easily specified in our logic. For this reason, we have extended the original version of LTL_{ss} with a binding operator inspired by a similar construction from TraceContract (Barringer and Havelund 2011), which provides a form of universal quantification over letters that makes it easy to define liveness properties, as we will explain in Section 3.3. Note that 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 (Nilsson 2014; Venners 2015).

Formulae Syntax We assume a denumerable set $\mathcal{V}$ of variables $(x, y, z, o, \ldots)$, a denumerable set $\mathcal{P}$ of predicate symbols $(p, q, r, \ldots)$ with associated arity – with $\mathcal{P}^n$ the set of predicate symbols with arity n, and $\mathbb{N} \subseteq \mathcal{F}^0$ –, and a denumerable set $\mathcal{F}$ of function

https://doi.org/10.1017/S1471068419000012 Published online by Cambridge University Press

symbols $(f, g, h, \ldots)$ with associated arity – with $\mathcal{F}^n$ the set of function symbols with arity n. Then, terms $e \in \mathit{Term}$ are built as

$$\mathit{Term} \ni e ::= \quad x \mid f(e_1, \ldots, e_n) \text{ for } x \in \mathcal{V}, f \in \mathcal{F}^n, e_1, \ldots, e_n \in \mathit{Term}$$

Typically, propositional formulations of LTL (Alur and Henzinger 1994) consider words that use the power set of atomic propositions as its alphabet. However, we consider the alphabet $\Sigma = \mathit{Term} \times \mathbb{N}$ of timed terms. Over this alphabet, we define the set of finite words Σ^*, that is, finite sequences of timed terms. We use ϵ for the empty word, and the notation $u = u_1 \ldots u_n$ to denote that $u \in \Sigma^*$ has length $\mathit{len}(u)$ equal to n, and u_i is the letter at position i in u. Each letter $u_i \equiv (e_i, t_i)$ corresponds to the term e_i that can be observed at instant i after t_i units of time have been elapsed. For example, for a Spark Streaming program with one input DStream and one output DStream, the term e_i would correspond to a pair of RDDs, one representing the microbatch for the input DStream at time t_i and another the microbatch for the output DStream at time t_i.

It is important to distinguish between the instant i, which refers to logic time and can be understood as a "counter of states," and t_i, which refers to real time. This real time satisfies the usual condition of *monotonicity* $(t_i \leq t_{i+1}, i \geq 0)$ but does not satisfy *progress* $(\forall t \in \mathbb{N}, \exists_{i \geq 0} t_i > t)$, since we work with finite words. It is also important to note that time is discrete but the time between successive states may be arbitrary. Also note that by the condition $\mathbb{N} \subseteq \mathcal{F}^0$, time literals are also terms, and therefore we can replace variables by terms and still obtain a term, as we will do later on in this section when defining the semantics of formulas.

The set of LTL_{ss} formulas LTL_{ss} is defined as follows:

$$\begin{aligned} LTL_{ss} \ni \varphi ::= \quad & \bot \mid \top \mid p(e_1, \ldots, e_n) \mid e_1 = e_2 \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \rightarrow \varphi_2 \mid \\ & X\varphi \mid \Diamond_t \varphi \mid \Box_t \varphi \mid \varphi \; U_t \; \varphi \mid \varphi \; R_t \; \varphi \mid \lambda^o_x.\varphi \end{aligned}$$

We will use the notation $X^n\varphi, n \in \mathbb{N}^+$, as a shortcut for n applications of the operator X to φ. Although, we provide a precise formulation for the interpretation of these formulas later in this section, the underlying intuitions are as follows:

- The first eight formulas are based on classical first-order non-temporal logical connectives, including contradiction, tautology, atomic formulas based on predicate application and equality, and the negation and the usual binary connectives.
- $X\varphi$, read "next φ," indicates that the formula φ should hold 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 eventually. 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.
- $\lambda^o_x.\varphi$, read "consume the current letter to produce φ," indicates that given (e, t) the letter for the current state, then the formula resulting from replacing in φ the variables x and o by e and t, respectively, should hold in the next state. We call this the *consume* operator.

https://doi.org/10.1017/S1471068419000012 Published online by Cambridge University Press

We say that a formula is *timeless* when it does not contain any of the temporal logical connectives. An LTL_{ss} formula or term is closed or ground if it has no free variables. In our framework variables are only bound by λ^o_x, so in the following we will consider a function $fv(\varphi)$ that computes the free variables of φ, discarding the appearances of x and o from ψ when $\lambda^o_x\psi$ is found and collecting the rest of them, including those appearing in temporal connectives.

We will only consider closed formulas in the following. Moreover, we will use the notation $\varphi[b \mapsto v_1, r \mapsto v_2] \equiv (\varphi[b \mapsto v_1])[r \mapsto v_2]$ to indicate that b and r are substituted by v_1 and v_2, respectively. A detailed explanation on how to compute the free variables and how to apply substitutions is available in Riesco and Rodríguez-Hortalá (2018).

Logic for finite words In order to evaluate our formulas, we need a way to interpret the timed terms that we use as the alphabet. In line with classical formulations of first-order Boolean logic (Smullyan 1995), formulas are evaluated in the context of an *interpretation structure* $\mathcal{A}$, which is a pair (A, I) where A is a non-empty set that is used as the interpreting domain, and I is an interpretation function that assigns to each $f \in \mathcal{F}^n$ an interpreting function $I(f) : A^n \to A$, and to each $p \in \mathcal{P}^n$ an interpreting relation $I(p) \subseteq A^n$. These interpretations are naturally applied to closed terms by induction on the structure of terms as $[\![f(e_1, \ldots, e_n)]\!]^{\mathcal{A}} = I(f)([\![e_1]\!]^{\mathcal{A}}, \ldots, [\![e_1]\!]^{\mathcal{A}}) \in A$. Our logic proves *judgments* of the form $u, i \vDash^{\mathcal{A}} \varphi : v$ that state that considering the finite word $u \in \Sigma^*$ from the position of its ith letter, the formula $\varphi \in LTL_{ss}$ is evaluated to the truth value $v \in \{\top, \bot, ?\}$ under the interpretation $\mathcal{A}$. In other words, if we stand at the ith letter of u and start evaluating φ, moving forward in u one letter at a time as time progresses, and using $\mathcal{A}$ to interpret the terms that appear in the word and in the formula, we end up getting the truth value v. Note that in our judgments, the same interpretation structure holds "eternally" constant for all instants, while only one letter of u is occurring at each instant. This is modeling what happens during the testing of a stream processing system: the code that defines how the program reacts to its inputs is the same during the execution of the program – which is modeled by a constant interpretation structure –, while the inputs of the program and their corresponding output change with time – which is modeled by the sequence of letters that is the word. This is not able to model updates in the program code, but it is expressive enough to be used during unit and integration testing, where the program code is fixed. Note the predicate symbols used in the formula correspond to the assertions used in the tests (Torreborre 2014), whose meaning is also constant during the test execution. Judgments are defined by the following rules, where only the first rule that fits should be applied, and we assume $\mathcal{A} = (A, I)$:

$$u, i \vDash^{\mathcal{A}} v : v \quad \text{if } v \in \{\bot, \top\},$$

$$u, i \vDash^{\mathcal{A}} p(e_1, \ldots, e_n) : \begin{cases} \top & \text{if } ([\![e_1]\!]^{\mathcal{A}}, \ldots, [\![e_n]\!]^{\mathcal{A}}) \subseteq I(p) \\ \bot & \text{otherwise,} \end{cases}$$

$$u, i \vDash^{\mathcal{A}} e_1 = e_2 : \begin{cases} \top & \text{if } [\![e_1]\!]^{\mathcal{A}} = [\![e_2]\!]^{\mathcal{A}} \\ \bot & \text{otherwise,} \end{cases}$$

https://doi.org/10.1017/S1471068419000012 Published online by Cambridge University Press

$$u, i \vDash^{\mathcal{A}} \lambda^o_x.\varphi : \begin{cases} v & \text{if } i \leq len(u) \wedge u, i+1 \vDash^{\mathcal{A}} \varphi[x \mapsto e_i, o \mapsto t_i] : v \text{ for } u_i \equiv (e_i, t_i) \\ ? & \text{otherwise,} \end{cases}$$

$$u, i \vDash^{\mathcal{A}} X\varphi : v \quad \text{if } u, i+1 \vDash^{\mathcal{A}} \varphi : v,$$

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

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

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

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

We say $u \vDash^{\mathcal{A}} \varphi$ iff $u, 1 \vDash^{\mathcal{A}} \varphi : \top$. The intuition underlying these definitions 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. Note the consume operator (λ^o_x) is the only one that accesses the word directly, and that consume is equivalent to next applied to the corresponding formula at its body: for example, $0\ \epsilon, 1 \vDash^{\mathcal{A}} \lambda^o_x.x = 0 : v \iff 0\ \epsilon, 1 \vDash^{\mathcal{A}} X(0 = 0) : v$. It is trivial to check that timeless formulas—i.e. without temporal connectives—are always evaluated to one of the usual binary truth values $\top$ of $\bot$, and that timeless formulas are evaluated to the same truth value irrespective of the word u and the position i considered, even for $u \equiv \epsilon$ or $i > len(u)$. As a consequence, some temporal formulas are true even for words with a length smaller than the number of letters referred by the temporal connectives in the formula: for example, for any i and $\mathcal{A}$ we have $\epsilon, i \vDash^{\mathcal{A}} X\top : \top$ – *next* inspects the second letter, but the formula is true for the empty word because the body is trivially true –, $0\ 1\ \epsilon, i \vDash^{\mathcal{A}} \Box_{10}\ (0 == 0) : \top$ – this *always* refers to 10 letters, but it holds for a word with just 2 letters because the body is a tautology –, and similarly $0\ \epsilon, 1 \vDash^{\mathcal{A}} \lambda^o_x.(x = 0) : \top$ because $0\ \epsilon, 2 \vDash^{\mathcal{A}} 0 = 0 : \top$.

https://doi.org/10.1017/S1471068419000012 Published online by Cambridge University Press

The resulting logic gives some structure to letters and words, but it is not fully a first-order logic because it does not provide neither existential or universal quantifiers for words. The *consume* operator is somewhat a universal quantifier for letters, but can also be understood as a construct for parameter passing, like the binding operator from TraceContract (Barringer and Havelund 2011).

Let us consider some example judgments for simple formulas, to start tasting the flavor of this logic.

Example 2
Assume the set of constants $\{a, b, c\} \subseteq \mathcal{F}^0$, the set of variables $\{x, y, z, o, p, q\}$, and an interpretation structure $\mathcal{A} \equiv (A, I)$ for the initial model where $A = \mathcal{F}^0$ and $\forall f \in \mathcal{F}^0.\ I(f) = f$. Then for the word $u \equiv \boxed{(b,0)}\boxed{(b,2)}\boxed{(a,3)}\boxed{(a,6)}$, we can construct the following formulas:

- $u \vDash^{\mathcal{A}} \Diamond_4\ \lambda^o_x.x = c : \bot$, since c does not appear in the first four letters.
- $u \vDash^{\mathcal{A}} \Diamond_5\ \lambda^o_x.x = c : ?$, since we have consumed the word, c did not appear in those letters but the timeout has not expired.
- $u \vDash^{\mathcal{A}} \Box_5\ \lambda^o_x.(x = a \vee x = b) : ?$, since the property holds until the word is consumed, but the user required more steps.
- $u \vDash^{\mathcal{A}} \lambda^o_x.x = b\ U_2\ \lambda^p_y.y = a : \bot$, since a appears in the third letter, but the user wanted to check just the first two letters.
- $u \vDash^{\mathcal{A}} \lambda^o_x.x = b\ U_5\ \lambda^p_y.y = a : \top$, since a appears in the third letter and, before that, b appeared in all the letters.
- $u \vDash^{\mathcal{A}} \lambda^o_x.x = a\ R_2\ \lambda^p_y.y = b : \top$, since b appears in all the required letters.
- $u \vDash^{\mathcal{A}} \Box_3(\lambda^o_x.x = a) \rightarrow X(\lambda^p_y.y = a) : \top$, since the formula holds in the first three letters (note that the fourth letter is required, since the formula involves the next operator).
- $u \vDash^{\mathcal{A}} \Box_2(\lambda^o_x.x = b) \rightarrow (\Diamond_2 \lambda^p_y.y = a) : \bot$, since in the first letter we have b but we do not have a until the third letter.
- $u \vDash^{\mathcal{A}} (\lambda^o_x.x = b)\ U_2\ X(\lambda^p_y.y = a \wedge X\lambda^q_z.z = a) : \top$, since $X(\lambda^p_y.y = a \wedge X\lambda^q_z.z = a)$ holds in the second letter (i.e., $(\lambda^p_y.y = a \wedge X\lambda^q_z.z = a)$ holds in the third letter, which can be also understood as a appears in the third and fourth letters).
- $u \vDash^{\mathcal{A}} \lambda^o_x.\Box_{o+6}\ x = b : \top$, since the first letter is b and hence the equality is evaluated to $\top$.

By using functions with arity greater than 0 and predicate symbols, we can construct more complex formulas. For example, given $\mathbb{N} \subseteq \mathcal{F}^0, plus \in \mathcal{F}^2, leq \in \mathcal{P}^2$, and an interpretation structure (A, I) where $A = \mathbb{N}$, $\forall n \in \mathbb{N}.\ I(n) = n$, $I(plus)(x, y) = x + y$, $I(leq) = \{(x, y) \in \mathbb{N} \times \mathbb{N} \mid x \leq y\}$, then we have $\boxed{(0,0)}\boxed{(1,2)}\boxed{(2,3)} \vDash^{\mathcal{A}} \Diamond_2\ \lambda^{o_1}_x.\lambda^{o_2}_y.leq(5, plus(x, y) : \top$

For some examples in this paper, we will assume *the Spark interpretation structure* $\mathcal{A}^{\mathcal{S}}$, which captures the observable semantics of a Spark program, and where timestamps are interpreted as Unix timestamps as usual in Java. We will not provide a formalization of $\mathcal{A}^{\mathcal{S}}$, but the idea is that the prototype we present in Section 3 is intended to implement a procedure to prove judgments under the Spark interpretation structure. This interpretation uses the set of Scala expressions as the domain A, and assumes that letters

https://doi.org/10.1017/S1471068419000012 Published online by Cambridge University Press

are timed tuples of terms and that each input or output DStream has an assigned tuple index, so that each element of the tuple represents the microbatch at that instant for the corresponding DStream. This is expressive enough to express any Spark Streaming program, because the set of DStreams is fixed during the lifetime of a Spark Streaming application. Let us see some simple formulas, we can write with this logic and in our prototype.

Example 3
Assuming a Spark Streaming program with one input DStream and one output DStream, the formula φ_1 below expresses the requirement that the output DStream will always contain numbers greater than 0, for 10 batches. As we have one input and one output, $\mathcal{A}^{\mathcal{S}}$ uses timed letters in $\mathbb{U} \times \mathbb{N}$ for $\mathbb{U} = \mathbb{RDD} \times \mathbb{RDD}$ and $\mathbb{RDD}$ the set of all Spark RDDs:

$$I(allOutValuesGtZero) = \{(rdd_i, rdd_o) \in \mathbb{U} \mid \forall e \in rdd_o.\ e \in \mathbb{Z} \wedge e > 0\}$$
$$\varphi_1 = \Box_{10}\ \lambda^{o}_{rdds}.allOutValuesGtZero(rdds).$$

This formula can be written in our prototype as follows:

```
always(nowTime[(RDD[Int], RDD[Int])]{ (letter, time) =>
  letter._2 should foreachRecord { _ > 0}
}) during 10
```

The formula φ_2 below expresses that time always increases monotonically during 10 instants:

$$I(leq) = \{(o_1, o_2) \in \mathbb{N} \times \mathbb{N} \mid o_1 \leq o_2\},$$
$$\varphi_2 = \Box_9\ \lambda^{o_1}_{x_1}.\lambda^{o_2}_{x_2}.leq(o_1, o_2),$$

which we can express in our prototype as

```
always(nextTime[(RDD[Int], RDD[Int])]{ (letter, time) =>
  nowTime[U]{ (nextLetter, nextTime) =>
    time.millis <= nextTime.millis
  }
}) during 9
```

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

Decision procedure for LTL_{ss} Just like ScalaCheck (Nilsson 2014) and any other testing tool of the QuickCheck family (Claessen and Hughes 2011; Papadakis and Sagonas 2011), this decision procedure does not try to be complete for proving the veritative value of formulae, but just to be complete for failures, that is judgments to the truth value $\bot$. For this purpose, we define an abstract rewriting system for reductions $u \vDash^{\mathcal{A}} \varphi \rightsquigarrow^* v$ for v in the same domain as above. We write $u \vDash \varphi \rightsquigarrow^* v$ when the interpretation $\mathcal{A}$ is implied by the context. Given a letter $a \in \Sigma$, a word $u \in \Sigma^*$, a set of terms $e, e_1, \ldots, e_n \in Term$, a timeout $t \in \mathbb{N}^+$, and formulas $\varphi, \varphi_1, \varphi_2 \in LTL_{ss}$, we have the following rules[1]:

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

https://doi.org/10.1017/S1471068419000012 Published online by Cambridge University Press

1. Rules for $u \vDash^{\mathcal{A}} p(e_i)$:
   (1) $u \vDash^{\mathcal{A}} p(e_1, \ldots, e_n) \rightsquigarrow \top$ if $([\![e_1]\!]^{\mathcal{A}}, \ldots, [\![e_n]\!]^{\mathcal{A}}) \subseteq I(p)$
   (2) $u \vDash^{\mathcal{A}} p(e_1, \ldots, e_n) \rightsquigarrow \bot$ otherwise
2. Rules for $u \vDash e_1 = e_2$:
   (1) $u \vDash^{\mathcal{A}} e_1 = e_2 \rightsquigarrow [\![e_1]\!]^{\mathcal{A}} = [\![e_2]\!]^{\mathcal{A}}$
3. Rules for $u \vDash \lambda_x^o.\varphi$:
   (1) $\epsilon \vDash \lambda_x^o.\varphi \rightsquigarrow ?$
   (2) $(e,t)u \vDash \lambda_x^o.\varphi \rightsquigarrow u \vDash \varphi[x \mapsto e][o \mapsto t]$
4. Rules for $u \vDash X\ \varphi$:
   (1) $\epsilon \vDash X\ \varphi \rightsquigarrow \epsilon \vDash \varphi$
   (2) $au \vDash X\ \varphi \rightsquigarrow u \vDash \varphi$
5. Rules for $u \vDash \Diamond_t\ \varphi$:
   (1) $\epsilon \vDash \Diamond_t\ \varphi \rightsquigarrow \epsilon \vDash \varphi$
   (2) $u \vDash \Diamond_0\ \varphi \rightsquigarrow \bot$
   (3) $u \vDash \Diamond_t\ \varphi \rightsquigarrow \top$ if $u \vDash \varphi \rightsquigarrow^* \top$
   (4) $au \vDash \Diamond_t\ \varphi \rightsquigarrow u \vDash \Diamond_{t-1}\ \varphi$ if $au \vDash \varphi \rightsquigarrow^* \bot$
6. Rules for $u \vDash \Box_t\ \varphi$:
   (1) $\epsilon \vDash \Box_t\ \varphi \rightsquigarrow \epsilon \vDash \varphi$
   (2) $u \vDash \Box_0\ \varphi \rightsquigarrow \top$
   (3) $u \vDash \Box_t\ \varphi \rightsquigarrow \bot$ if $u \vDash \varphi \rightsquigarrow^* \bot$
   (4) $au \vDash \Box_t\ \varphi \rightsquigarrow u \vDash \Box_{t-1}\ \varphi$ if $au \vDash \varphi \rightsquigarrow^* \top$
7. Rules for $u \vDash \varphi_1\ U_t\ \varphi_2$:
   (1) $\epsilon \vDash \varphi_1\ U_t\ \varphi_2 \rightsquigarrow \epsilon \vDash \varphi_2$
   (2) $u \vDash \varphi_1\ U_0\ \varphi_2 \rightsquigarrow \bot$
   (3) $u \vDash \varphi_1\ U_t\ \varphi_2 \rightsquigarrow \top$ if $u \vDash \varphi_2 \rightsquigarrow^* \top$
   (4) $u \vDash \varphi_1\ U_t\ \varphi_2 \rightsquigarrow \bot$ if $u \vDash \varphi_1 \rightsquigarrow^* \bot \wedge u \vDash \varphi_2 \rightsquigarrow^* \bot$
   (5) $au \vDash \varphi_1\ U_t\ \varphi_2 \rightsquigarrow u \vDash \varphi_1\ U_{t-1}\ \varphi_2$ if $au \vDash \varphi_1 \rightsquigarrow^* \top \wedge au \vDash \varphi_2 \rightsquigarrow^* \bot$
8. Rules for $u \vDash \varphi_1\ R_t\ \varphi_2$:
   (1) $\epsilon \vDash \varphi_1\ R_t\ \varphi_2 \rightsquigarrow \epsilon \vDash \varphi_1$
   (2) $u \vDash \varphi_1\ R_0\ \varphi_2 \rightsquigarrow \top$
   (3) $u \vDash \varphi_1\ R_t\ \varphi_2 \rightsquigarrow \top$ if $u \vDash \varphi_1 \rightsquigarrow^* \top \wedge u \vDash \varphi_2 \rightsquigarrow^* \top$
   (4) $u \vDash \varphi_1\ R_t\ \varphi_2 \rightsquigarrow \bot$ if $u \vDash \varphi_2 \rightsquigarrow^* \bot$
   (5) $au \vDash \varphi_1\ R_t\ \varphi_2 \rightsquigarrow u \vDash \varphi_1\ R_{t-1}\ \varphi_2$ if $au \vDash \varphi_1 \rightsquigarrow^* \bot \wedge au \vDash \varphi_2 \rightsquigarrow^* \top$

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 continues 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, that is, what is happening now, but also the subsequent letters, as illustrated by the following examples:

Example 4
We recall the word $u \equiv \boxed{(b,0)}\ \boxed{(b,2)}\ \boxed{(a,3)}\ \boxed{(a,6)}$ from Example 2 and evaluate the following formulas:

https://doi.org/10.1017/S1471068419000012 Published online by Cambridge University Press

- $\boxed{(b,0)}\ \boxed{(b,2)}\ \boxed{(a,3)}\ \boxed{(a,6)} \vDash \Box_2(\lambda^o_x.x = b) \to (\Diamond_2\ \lambda^p_y.y = a) \rightsquigarrow \bot$, because first the x in $(\lambda^o_x.x = b)$ is bound to b and hence the premise holds, but $\boxed{(b,0)}\ \boxed{(b,2)}\ \boxed{(a,3)}\ \boxed{(a,6)} \vDash (\Diamond_2\lambda^p_y.y = a) \rightsquigarrow$
$\boxed{(b,2)}\ \boxed{(a,3)}\ \boxed{(a,6)} \vDash (\Diamond_1\ \lambda^p_y.y = a) \rightsquigarrow$
$\boxed{(a,3)}\ \boxed{(a,6)} \vDash (\Diamond_0\lambda^p_y.y = a) \rightsquigarrow \bot$.
- $\boxed{(b,0)}\ \boxed{(b,2)}\ \boxed{(a,3)}\ \boxed{(a,6)} \vDash (\lambda^o_x.x = b)\ U_2\ X(\lambda^p_y.y = a \wedge X\lambda^q_z.z = a) \rightsquigarrow$
$\boxed{(b,2)}\ \boxed{(a,3)}\ \boxed{(a,6)} \vDash (\lambda^o_x.x = b)\ U_1\ X(\lambda^p_y.y = a \wedge X\lambda^q_z.z = a)$, which requires to check the second and third letters to check that the second formula does not hold. Then we have $\boxed{(b,2)}\ \boxed{(a,3)}\ \boxed{(a,6)} \vDash (\lambda^o_x.x = b)\ U_1\ X(\lambda^p_y.y = a \wedge X\lambda^q_z.z = a) \rightsquigarrow \top$ after checking the third and fourth letters.
- $\boxed{(b,0)}\ \boxed{(b,2)}\ \boxed{(a,3)}\ \boxed{(a,6)} \vDash \lambda^o_x.\Box_{o+6}x = b \rightsquigarrow$
$\boxed{(b,2)}\ \boxed{(a,3)}\ \boxed{(a,6)} \vDash \Box_6\top$, just by binding the variables. Then we have
$\boxed{(b,2)}\ \boxed{(a,3)}\ \boxed{(a,6)} \vDash \Box_6\top \rightsquigarrow$
$\boxed{(a,3)}\ \boxed{(a,6)} \vDash \Box_5\top \rightsquigarrow$
$\boxed{(a,6)} \vDash \Box_4\top \rightsquigarrow \varepsilon \vDash \Box_3\top \rightsquigarrow \varepsilon \vDash \top \rightsquigarrow \top$ just by applying the rules for $\Box$.

To use this procedure as the basis for our implementation, we would have to keep a list of suspended alternatives for each of the rules above, that are pending the resolution of the conditions that define each alternative, which will be solved in the future. For example, if we apply rule 5 to an application of $\Diamond_t$ for a non-empty word and $t > 0$ then we get two alternatives for sub-rules 5.3 and 5.4, and those alternatives will depend on whether the nested formula φ is reduced to $\top$ or $\bot$ using $\rightsquigarrow^*$, which cannot be determined at the instant when rule 5 is applied. This is because, although we do have 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 instants, as our implementation runs the actual code that is the subject of the test in a local Spark cluster. 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 inspect 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 (i.e., 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 operators next and consume, which leads us to the following definition.

Definition 2.1 (*Next form*)
We say that a formula $\psi \in LTL_{ss}$ is in *next form* iff. it is built by using the following grammar:

$$\psi ::= \quad \bot \mid \top \mid p(e,\ldots,e) \mid e = e \mid \neg\psi \mid \psi \vee \psi \mid \psi \wedge \psi \mid \psi \to \psi \mid X\psi \mid \lambda^o_x.\psi.$$

We can extend the transformation in Riesco and Rodríguez-Hortalá (2016b) for computing the next form of any formula $\varphi \in LTL_{ss}$:

https://doi.org/10.1017/S1471068419000012 Published online by Cambridge University Press

Definition 2.2 (*Explicit next transformation*)
Given a formula $\varphi \in LTL_{ss}$, the function $nt^e(\varphi)$ computes another formula $\varphi' \in LTL_{ss}$, such that φ' is in *next form* and $\forall u \in \Sigma^*.u \vDash \varphi \iff u \vDash \varphi'$.

$$
\begin{array}{lcll}
nt^e(\varphi) & = & \varphi & \text{if } \varphi \in \{\top, \bot, p(e_1, \ldots, e_n), e_1 = e_2\} \\
nt^e(op\ \varphi) & = & op\ nt^e(\varphi) & \text{if } op \in \{\neg, X, \lambda_x^o\} \\
nt^e(\varphi_1\ op\ \varphi_2) & = & nt^e(\varphi_1)\ op\ nt^e(\varphi_2) & \text{if } op \in \{\vee, \wedge, \rightarrow\} \\
nt^e(\Diamond_t \varphi) & = & nt^e(\varphi) \vee X nt^e(\varphi) \vee \ldots \vee X^{t-1} nt^e(\varphi) & \\
nt^e(\Box_t \varphi) & = & nt^e(\varphi) \wedge X nt^e(\varphi) \wedge \ldots \wedge X^{t-1} nt^e(\varphi) & \\
nt^e(\varphi_1\ U_t\ \varphi_2) & = & nt^e(\varphi_2) \vee (nt^e(\varphi_1) \wedge X nt^e(\varphi_2)) \vee & \\
 & & (nt^e(\varphi_1) \wedge X nt^e(\varphi_1) \wedge X^2 nt^e(\varphi_2)) \vee \ldots \vee & \\
 & & (nt^e(\varphi_1) \wedge X nt^e(\varphi_1) \wedge \ldots \wedge X^{t-2} nt^e(\varphi_1) \wedge X^{t-1} nt^e(\varphi_2)) & \\
nt^e(\varphi_1\ R_t\ \varphi_2) & = & (nt^e(\varphi_2) \wedge X nt^e(\varphi_2) \wedge \ldots \wedge X^{t-1} nt^e(\varphi_2)) \vee & \\
 & & (nt^e(\varphi_1) \wedge nt^e(\varphi_2)) \vee (nt^e(\varphi_2) \wedge X(nt^e(\varphi_1) \wedge nt^e(\varphi_2))) \vee & \\
 & & (nt^e(\varphi_2) \wedge X nt^e(\varphi_2) \wedge X^2(nt^e(\varphi_1) \wedge nt^e(\varphi_2))) \vee \ldots \vee & \\
 & & (nt^e(\varphi_2) \wedge X nt^e(\varphi_2) \wedge \ldots \wedge X^{t-2} nt^e(\varphi_2) \wedge X^{t-1}(nt^e(\varphi_1) \wedge nt^e(\varphi_2)) &
\end{array}
$$

for $e_1, e_2 \in Term$, $x, o \in \mathcal{V}$, $p \in \mathcal{P}^n$, and $\varphi, \varphi_1, \varphi_2 \in LTL_{ss}$.

Note that (i) it is not always possible to compute the next form a priori, since the time in temporal operators might contain variables that need to be bound and (ii) the transformation might produce large formulas. For these reasons, it is worth transforming the formula following a lazy strategy, which only generates the subformulas required in the current and the next states. We present next a recursive function that allows us to compute the next form in a lazy way, which we use to improve the efficiency of our prototype, as we will see in Section 3.1.

Definition 2.3 (*Recursive next transformation*)
Given a formula $\varphi \in LTL_{ss}$, the function $nt(\varphi)$ computes another formula $\varphi' \in LTL_{ss}$, such that φ' is in *next form* and $\forall u \in \Sigma^*.u \vDash \varphi \iff u \vDash \varphi'$:

$$
\begin{array}{lcll}
nt(\varphi) & = & \varphi & \text{if } \varphi \in \{\top, \bot, p(e_1, \ldots, e_n), e_1 = e_2\} \\
nt(op\ \varphi) & = & op\ nt(\varphi) & \text{if } op \in \{\neg, X, \lambda_x^o\} \\
nt(\varphi_1\ op\ \varphi_2) & = & nt(\varphi_1)\ op\ nt(\varphi_2) & \text{if } op \in \{\vee, \wedge, \rightarrow\} \\
nt(\Diamond_1 \varphi) & = & nt(\varphi) & \\
nt(\Diamond_t \varphi) & = & nt(\varphi) \vee X nt(\Diamond_{t-1} \varphi) & \text{if } t \geq 2 \\
nt(\Box_1 \varphi) & = & nt(\varphi) & \\
nt(\Box_t \varphi) & = & nt(\varphi) \wedge X nt(\Box_{t-1} \varphi) & \text{if } t \geq 2 \\
nt(\varphi_1\ U_1\ \varphi_2) & = & nt(\varphi_2) & \\
nt(\varphi_1\ U_t\ \varphi_2) & = & nt(\varphi_2) \vee & \\
 & & (nt(\varphi_1) \wedge X nt(\varphi_1\ U_{t-1}\ \varphi_2)) & \text{if } t \geq 2 \\
nt(\varphi_1\ R_1\ \varphi_2) & = & nt(\varphi_1) \wedge nt(\varphi_2) & \\
nt(\varphi_1\ R_t\ \varphi_2) & = & (nt(\varphi_1) \wedge nt(\varphi_2)) \vee & \\
 & & (nt(\varphi_2) \wedge X nt(\varphi_1\ R_{t-1}\ \varphi_2)) & \text{if } t \geq 2
\end{array}
$$

for $e_1, \ldots e_n \in Term$, $x, o \in \mathcal{V}$, $p \in \mathcal{P}^n$, and $\varphi, \varphi_1, \varphi_2 \in LTL_{ss}$.

Next, we present some results about these transformations and an auxiliary lemma that indicates that, if two formulas are equivalent at time 1, then they keep being equivalent the rest of the execution:

https://doi.org/10.1017/S1471068419000012 Published online by Cambridge University Press

Lemma 2.1
Given an alphabet Σ and formulas $\varphi, \varphi' \in LTL_{ss}$, if $\forall u \in \Sigma^*.u, 1 \vDash \varphi \iff u, 1 \vDash \varphi'$ then $\forall u \in \Sigma^*, \forall n \in \mathbb{N}^+.u, n \vDash \varphi \iff u, n \vDash \varphi'$.

Theorem 1 (*Transformation equivalence*)
Given a formula $\varphi \in LTL_{ss}$ such that φ does not contain variables in temporal connectives, we have $nt(\varphi) = nt^e(\varphi)$.

It is straightforward to see that the formula obtained by this transformation is in *next form*, since it only introduces formulas using the temporal operators next or consume. The equivalence between formulas is stated in Theorem 2:

Theorem 2
Given an alphabet Σ, an interpretation $\mathcal{A}$, and formulas $\varphi, \varphi' \in LTL_{ss}$, such that $\varphi' \equiv nt(\varphi)$, we have $\forall u \in (\Sigma \times \mathbb{N})^*.u \vDash^{\mathcal{A}} \varphi \iff u \vDash^{\mathcal{A}} \varphi'$.

Both theorems are proved by induction in the structure of formulas and using Lemma 2.1. Detailed proofs are available in Appendix A in the supplementary material.

The next section show some examples of explicit transformation and the first step of the lazy transformation.

Example 5
We present here how to transform some of the formulas from Example 2. Note that the last one cannot be completely transformed a priori:

- $nt^e(\Diamond_4\ \lambda^o_x.x = c) = \lambda^o_x.x = c \vee X\lambda^o_x.x = c \vee X^2\lambda^o_x.x = c \vee X^3\lambda^o_x.x = c$
- $nt^e(\lambda^o_x.x = b\ U_2\ \lambda^o_x.x = a) = \lambda^o_x.x = a \vee (\lambda^o_x.x = b \wedge X\lambda^o_x.x = a)$
- $nt^e(\Box_2(\lambda^o_x.x = b) \rightarrow (\Diamond_2\lambda^p_y.y = a) = (\lambda^o_x.x = b \rightarrow (\lambda^p_y.y = a \vee X\lambda^p_y.y = a)) \wedge X(\lambda^o_x.x = b \rightarrow (\lambda^p_y.y = a \vee X\lambda^p_y.y = a))$
- $nt^e((\lambda^o_x.x = b)\ U_2\ X(\lambda^p_y.y = a \wedge X\lambda^q_z.z = a)) = X(\lambda^p_y.y = a \wedge X\lambda^q_z.z = a) \vee (\lambda^o_x.x = b \wedge X^2(\lambda^p_y.y = a \wedge X\lambda^q_z.z = a))$.

Example 6
We present the lazy next transformation for some formulas, where we just apply the first transformation. Note that in the last example, it is not possible to compute the next form in an eager way:

- $nt(\Box_2(\lambda^o_x.x = b) \rightarrow (\Diamond_2\lambda^p_y.y = a)) = (\lambda^o_x.x = b) \rightarrow (\Diamond_2\lambda^p_y.y = a) \wedge X\,nt(\Box_1(\lambda^o_x.x = b) \rightarrow (\Diamond_2\lambda^p_y.y = a))$
- $nt(\lambda^o_x.\Box_{o+6}x = b) = \lambda^o_x.nt(\Box_{o+6}x = b)$

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 (i.e., 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 2.4 (*Letter simplification*)
Given a formula ψ in next form, a letter $s \in \Sigma$, where s can be either (e, t), with $e \in Term, t \in \mathbb{N}$, or the empty letter $\emptyset$, and an interpretation $\mathcal{A} = (A, I)$, the

https://doi.org/10.1017/S1471068419000012 Published online by Cambridge University Press

function $ls^{\mathcal{A}}(\psi, s)$ ($ls(\psi, s)$ when $\mathcal{A}$ is clear from the context) *simplifies* ψ *with* s as follows:

$$
\begin{array}{lcll}
ls(\varphi, s) & = & \varphi & \text{if } \varphi \in \{\top, \bot\} \\
ls^{\mathcal{A}}(p(e_1, \ldots, e_n), s) & = & ([\![e_1]\!]^{\mathcal{A}}, \ldots, [\![e_n]\!]^{\mathcal{A}}) \subseteq I(p) & \\
ls^{\mathcal{A}}(e_1 = e_2, s) & = & synEq([\![e_1]\!]^{\mathcal{A}}, [\![e_2]\!]^{\mathcal{A}}) & \\
ls(\psi_1 \ op \ \psi_2, s) & = & ls(\psi_1, s) \ op \ ls(\psi_2, s) & \text{if } op \in \{\vee, \wedge, \rightarrow\} \\
ls(X\psi, (e, t)) & = & \psi & \\
ls(X\psi, \emptyset) & = & ls(\psi, \emptyset) & \\
ls(\lambda^o_x.\psi, (e, t)) & = & \psi[x \mapsto e][o \mapsto t] & \\
ls(\lambda^o_x.\psi, \emptyset) & = & ?, &
\end{array}
$$

where *synEq* stands for syntactic equality. Note that using the empty letter forces the complete evaluation of the formula. Using this function and applying propositional logic and the interpretation $\mathcal{A}$ 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 illustrated in the next example.

Example 7

We present here the lazy evaluation process for some formulas in Example 4 using the word $u \equiv \boxed{(b,0)}\boxed{(b,2)}\boxed{(a,3)}\boxed{(a,6)}$.

- $nt(\Box_2(\lambda^o_x.x = b) \rightarrow (\Diamond_2\lambda^p_y.y = a)) = (\lambda^o_x.x = b) \rightarrow (\lambda^p_y.y = a \vee (X nt(\Diamond_1\lambda^p_y.y = a))) \wedge X nt(\Box_1(\lambda^o_x.x = b) \rightarrow (\Diamond_2\lambda^p_y.y = a))$ (from Example 6).
  - $ls((\lambda^o_x.x = b) \rightarrow (\lambda^p_y.y = a \vee (X nt(\Diamond_1\lambda^p_y.y = a))) \wedge X nt(\Box_1(\lambda^o_x.x = b) \rightarrow (\Diamond_2\lambda^p_y.y = a)), (b,0))) =$ (consume letter)
    $\top \rightarrow (\bot \vee nt(\Diamond_1\lambda^p_y.y = a)) \wedge nt(\Box_1(\lambda^o_x.x = b) \rightarrow (\Diamond_2\lambda^p_y.y = a))$
    $\equiv$ (simplification)
    $nt(\Diamond_1\lambda^p_y.y = a) \wedge nt(\Box_1(\lambda^o_x.x = b) \rightarrow (\Diamond_2\lambda^p_y.y = a))$
    $\equiv$ (lazy evaluation of the next transformation)
    $\lambda^p_y.y = a \wedge \lambda^o_x.x = b \rightarrow (\lambda^p_y.y = a \vee X nt(\Diamond_1\lambda^p_y.y = a)))$.
  - $ls(\lambda^p_y.y = a \wedge \lambda^o_x.x = b \rightarrow (\lambda^p_y.y = a \vee X nt(\Diamond_1\lambda^p_y.y = a))), (b,2))$
    $=$ (consume letter)
    $\bot \wedge \top \rightarrow (\bot \vee nt(\Diamond_1\lambda^p_y.y = a)))$
    $\equiv$ (simplification)
    $\bot$.
- $nt(\lambda^o_x.\Box_{o+6}x = b) = \lambda^o_x.nt(\Box_{o+6}x = b)$ (from Example 6).
  - $ls(\lambda^o_x.nt(\Box_{o+6}x = b), (b,0)) = nt(\Box_6 b = b)$
    $\equiv$ (lazy evaluation of the next transformation)
    $\top \wedge X nt(\Box_5 b = b)$
    $\equiv$ (simplification)
    $X nt(\Box_5 b = b)$

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

https://doi.org/10.1017/S1471068419000012 Published online by Cambridge University Press

— $ls(X\,nt(\Box_5 b = b), (b, 2)) = nt(\Box_5 b = b)$
$\equiv$ (lazy evaluation of the next transformation and simplification)
$X\,nt(\Box_4 b = b)$
— $ls(X\,nt(\Box_4 b = b), (a, 3)) = nt(\Box_4 b = b)$
$\equiv$ (lazy evaluation of the next transformation and simplification)
$X\,nt(\Box_3 b = b)$
— $ls(X\,nt(\Box_3 b = b), (a, 6)) = nt(\Box_3 b = b)$
$\equiv$ (lazy evaluation of the next transformation and simplification)
$X\,nt(\Box_2 b = b)$
— $ls(X\,nt(\Box_3 b = b), \emptyset) = nt(\Box_3 b = b)$
$\equiv$ (lazy evaluation of the next transformation and simplification)
$X\,nt(\Box_2 b = b)$
— $ls(X\,nt(\Box_2 b = b), \emptyset) = ls(nt(\Box_2 b = b), \emptyset)$
$\equiv$ (lazy evaluation of the next transformation and simplification)
$ls(nt(\Box_1 b = b), \emptyset)$
$\equiv$ (lazy evaluation of the next transformation and simplification)
$ls(\top, \emptyset) \equiv \top$

When no variables appear in the timeouts of temporal operators, 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/consume operators nested in the transformation.[3] We define this *safe word length* as follows:

Definition 2.5 (*Safe word length*)
Given a formula $\varphi \in LTL_{ss}$ without variables in any timeouts of the temporal operators that occur in it, 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:

$$
\begin{array}{lcll}
swl(\varphi) & = & 0 & \text{if } \varphi \in \{\top, \bot, p(e_1, \ldots, e_n), e_1 = e_2\} \\
swl(\neg\varphi) & = & swl(\varphi) & \\
swl(\varphi_1\ op\ \varphi_2) & = & max(swl(\varphi_1), swl(\varphi_2)) & \text{if } op \in \{\vee, \wedge, \rightarrow\} \\
swl(op\ \varphi) & = & swl(\varphi) + 1 & \text{if } op \in \{X, \lambda^o_x\} \\
swl(op_t\ \varphi) & = & swl(\varphi) + (t - 1) & \text{if } op \in \{\Diamond, \Box\} \\
swl(\varphi_1\ U_t\ \varphi_2) & = & max(swl(\varphi_1), swl(\varphi_2)) + (t - 1) & \text{if } op \in \{U, R\}
\end{array}
$$

Example 8
We present here the safe word length for some of the formulas in Example 2:

- $swl(\Box_3(\lambda^o_x.x = a) \rightarrow X(\lambda^p_y.y = a)) = 4$.
- $swl((\lambda^o_x.x = b)\ U_2\ X(\lambda^p_y.y = a \wedge X\lambda^q_z.z = a)) = 4$.

On the other hand, we cannot define a safe word length for arbitrary formulas with variables in timeouts, because an application of the consume operator might bind those variables using a letter of the input word, so there is no way to determine the value of the timeout for all possible words.

[3] Note that it might be possible to avoid an inconclusive value with shorter words, so this is a *sufficient* condition.

https://doi.org/10.1017/S1471068419000012 Published online by Cambridge University Press

2.3 Generating words

Besides stating properties, formulas can be used to generate words. In particular, we will generate sequences of terms from formulas; these sequences can then be extended by pairing each letter with a number generated by an arbitrary monotonically increasing function, hence obtaining words with timed terms as letters. The formulas used for generating terms have the following restrictions:

- Given a formula $\lambda_x^o.\varphi$, we have $o \notin fv(\varphi)$. Since, in this stage, we do not generate times, they cannot be used.
- Formulas do not contain the negation operator or the `false` constant. The process tries to generate words that make the formula evaluate to true, so we would not generate any word for a contradiction. Besides, we do not support negation because that would imply maintaining a set of excluded words, and we wanted to define simple ScalaCheck generators in a straightforward way.

For describing how the generators compute the sequences we first need to introduce a constant $err \in Term^*$ that stands for an erroneous sequence. Moreover, we use the notation $+ : Term^* \times Term^* \to Term^*$ $(u + err = err + u = err)$ for composing words, and extend the union on $Term^*$ as

$$\begin{array}{lcl} a\ u \cup b\ v & = & (a \cup b) + u \cup v \\ u \cup \epsilon & = & u \\ u \cup err & = & err \end{array}$$

for $a, b \in Term$ and $u, v \in Term^*$. Note that we assume that syntax for sets and unions is defined in $\mathcal{F}$. Using these ideas, we have

Definition 2.6 (*Random word generation*)
Given an interpretation $\mathcal{A}$, $e_1, \ldots, e_n \in Term$, $p \in \mathcal{P}^n$, formulas ψ, ψ_1, and ψ_2 in next form, the function $gen^{\mathcal{A}}$ (*gen* when $\mathcal{A}$ is clear from the context) generates a finite word $u \in Term^*$. If different equations can be applied for a given formula any of them can be chosen:

$$\begin{array}{lcll} gen(\top) & = & \emptyset & \\ gen^{\mathcal{A}}(p(e_1, \ldots, e_n)) & = & \emptyset & \text{if } ([\![e_1]\!]^{\mathcal{A}}, \ldots, [\![e_n]\!]^{\mathcal{A}}) \subseteq I(p) \\ gen^{\mathcal{A}}(e_1 = e_2) & = & \emptyset & \text{if } [\![e_1]\!]^{\mathcal{A}} = [\![e_2]\!]^{\mathcal{A}} \\ gen(\psi_1 \vee \psi_2) & = & gen(\psi_1) & \\ gen(\psi_1 \vee \psi_2) & = & gen(\psi_2) & \\ gen(\psi_1 \wedge \psi_2) & = & gen(\psi_1) \cup gen(\psi_2) & \\ gen(\psi_1 \to \psi_2) & = & gen(\psi_2) & \\ gen(X\psi) & = & \emptyset + gen(\psi) & \\ gen(\lambda_x^o.\psi) & = & \{e\} + gen(\psi) & \text{if } x \notin fv(\psi), \text{ pick an } e \in Term \\ & & & \text{s.t. } gen(\psi[x \mapsto e]) \neq err \\ gen(\psi) & = & err & \text{otherwise,} \end{array}$$

where $\emptyset$ stands for an empty term and indicates that the batch can be empty.

Note that this definition interprets conjunctions as unions. Hence, the formula $\psi \equiv (\lambda_x^o.x = a) \wedge (\lambda_x^o.x = b)$ is interpreted as $\psi \equiv (\lambda_x^o.x \supset \{a\}) \wedge (\lambda_x^o.x \supset \{b\})$ and generates a single batch containing a and b.

https://doi.org/10.1017/S1471068419000012 Published online by Cambridge University Press

Example 9
We present here the generation process for a formula from Example 2:

- $gen(\Box_2(\lambda^o_x.x = b) \rightarrow (\Diamond_2\lambda^p_y.y = a)) = gen((\lambda^o_x.x = b) \rightarrow (\lambda^p_y.y = a \vee (Xnt(\Diamond_1\lambda^p_y.y = a))) \wedge Xnt(\Box_1(\lambda^o_x.x = b) \rightarrow (\Diamond_2\lambda^p_y.y = a)))$ (from Example 6).
- $gen((\lambda^o_x.x = b) \rightarrow (\lambda^p_y.y = a \vee (Xnt(\Diamond_1\lambda^p_y.y = a))) \wedge Xnt(\Box_1(\lambda^o_x.x = b) \rightarrow (\Diamond_2\lambda^p_y.y = a))) =$
  $gen((\lambda^o_x.x = b) \rightarrow (\lambda^p_y.y = a \vee (Xnt(\Diamond_1\lambda^p_y.y = a)))) \cup$
  $gen(Xnt(\Box_1(\lambda^o_x.x = b) \rightarrow (\Diamond_2\lambda^p_y.y = a))) =$
  $\boxed{a} \cup \boxed{\emptyset}\ \boxed{a} = \boxed{a}\ \boxed{a}$.

Since we have, for the first term of the union:

- $gen((\lambda^o_x.x = b) \rightarrow (\lambda^p_y.y = a \vee (Xnt(\Diamond_1\lambda^p_y.y = a)))) =$
  $gen(\lambda^p_y.y = a \vee (Xnt(\Diamond_1\lambda^p_y.y = a))) =$
  $gen(\lambda^p_y.y = a) =$
  $\boxed{a}$.

Similarly, we would generate the second term of the union. Note that in both cases, we decided to generate values for the first term of the disjunction. A similar process can be followed to obtain different values.

3 sscheck: using LTL_{ss} for property-based testing

We have developed a prototype that allows for using the LTL_{ss} logic for property-based testing of Spark Streaming programs, as the Scala library sscheck (Riesco and Rodríguez-Hortalá 2017b). This library extends the PBT library ScalaCheck (Nilsson 2014) with custom generators for Spark DStreams and with a property factory that allows developers to check an LTL_{ss} formula against the finite DStream prefixes generated by another LTL_{ss} formula.

3.1 Design overview

In order to write a temporal property in sscheck, the user extends the trait (the Scala version of an abstract class) `DStreamTLProperty` and then implements some abstract methods to configure Spark Streaming (e.g., defining the batch interval or the Spark master). The method `DStreamTLProperty.forAllDStream` is used to define temporal ScalaCheck properties:

```
type SSeq[A] = Seq[Seq[A]]
type SSGen[A] = Gen[SSeq[A]]

def forAllDStream[In:ClassTag,Out:ClassTag](
  generator: SSGen[In])(
  transformation: (DStream[In]) => DStream[Out])(
  formula: Formula[(RDD[In], RDD[Out])])(
  implicit pp1: SSeq[In] => Pretty): Prop
```

The function `forAllDStream` takes a ScalaCheck generator of sequences of sequences of elements, which are interpreted as finite DStream prefixes, so each nested

https://doi.org/10.1017/S1471068419000012 Published online by Cambridge University Press

sequence is interpreted as an RDD. Our library defines a case class `Batch[A]` that extends `Seq[A]` to represent an RDD for a microbatch, and a case class `PDStream[A]` that extends `Seq[Batch[A]]` to represent a finite DStream prefix. For example, `Batch("scala", "spark")` represents an `RDD[String]` with two elements, and `PDStream(Batch("scala", "spark"), Batch(), Batch("spark"))` represents a finite prefix of a `DStream[String]` consisting of a microbatch with two elements, followed by an empty microbatch, and finally a microbatch with a single element. The sscheck classes `BatchGen` and `PDStreamGen` and their companion objects can be used to define generators of `Batch` and `PDStream` objects using temporal operators, and the trait `Formula` is used to represent LTL_{ss} formulas. See Section 3.2 below for details about the user API to write properties with sscheck. Note the type parameter of `Formula` is `(RDD[In], RDD[Out])`, which means in `formula` the letter corresponding to each instant is a pair of RDDs, one for the input DStream and another for the output DStream. Finally, the function `transformation` is the *test subject* which correctness is checked during the evaluation of the property.

In order to evaluate the resulting ScalaCheck `Prop`, first we apply a lazy variant of the transformation from Definition 2.3 (see Section 3.2 for details.) to `formula`, in order to get an equivalent formula in next form. Then the following process iterates until the specified number of test cases has passed, or until a failing test case – that is, a counterexample – is found, whatever happens first. A test case of type `SSeq[In]` is generated using `generator`, which corresponds to a finite prefix for the input DStream, and a fresh Spark `StreamingContext` is created. The test case, the streaming context, and the transformation are used to create a `TestCaseContext` that encapsulates the execution of the test case. The program then blocks until the test case is executed completely by the Spark runtime, and then a result for the test case is returned by the test case context. Test case results can be inconclusive, which correspond to the truth value ? in LTL_{ss}; in case the generated test case is too short for the formula. Internally, the test case context defines an input DStream by parallelizing the test case – using the Spark-testing-base package (Karau 2015b) – and applies the test subject `transformation` to it to define an output DStream. It also maintains variables for the number of remaining batches (initialized to the length of the test case), and the current value for the formula, and registers a `foreachRDD` Spark action that updates the number of remaining batches, and the current formula using the letter simplification procedure from Definition 2.4. This action also stops the Spark streaming context once the formula is solved or there are no remaining batches. Other variants of `forAllDStream` can be used for defining properties with more than one input DStream and one output DStream.

Therefore, `forAllDStream(gen)(transformation)(formula)` is trying to refute $\forall u_g \in gen(\varphi_g).(u, 1 \vDash^{\mathcal{A}^S} \varphi_p : \top) \vee (u, 1 \vDash^{\mathcal{A}^S} \varphi_p : ?)$ for the Spark interpretation structure $\mathcal{A}^S$, formulas φ_g, φ_p corresponding to `gen` and `formula` respectively, and $u \equiv zip(zip(u_g, u_o), u_t)$ where u_o is a word which interpretation under $\mathcal{A}^S$ corresponds to the result of applying `transformation` to the interpretation of u_g under $\mathcal{A}^S$ and $u_t = ct\ (ct + b)\ (ct + 2b)\ (ct + 3b) \ldots$ is the sequence of time stamps starting from the unix timestamp ct at the start of the execution of the property and moving b milliseconds at a time for b the configured batch interval. Here zip is the usual operator that combines two sequences element wise to produce a sequence of pairs of elements in the

https://doi.org/10.1017/S1471068419000012 Published online by Cambridge University Press

same position, truncating the longest of the two sequences to the length of the shortest. This way, we add an additional external universal quantifier on the domain of finite words, as usual in PBT, but inside that scope we have a propositional LTL_{ss} formula, and we evaluate the whole formula with the usual sound but incomplete PBT evaluation procedure.

3.2 User manual

In order to check the behavior of a test subject `transformation` the user defines a property using LTL_{ss} logic by invoking the method `forAllDStream[In, Out]` with the following arguments:

Generator `gen: Gen[Seq[Seq[In]]].` It is a regular ScalaCheck generator that produces sequences of sequences of elements, where each nested sequence represents an RDD for a Spark Streaming microbatch, and where the top sequence represent a prefix of a DStream. We represent that with the classes `Batch[A](points : A*) extends Seq[A]` for the batches, and `PDStream[A](batches : Batch[A]*) extends Seq[Batch[A]]` for the DStream prefixes. The objects `PDStreamGen` and `BatchGen` define functions for a small combinator library for ScalaCheck generators using the temporal operators of LTL_{ss}. First of all, `BatchGen.ofN[T](n: Int, g: Gen[T]): Gen[Batch[T]]` can be used to define a batch generator of batches of size n from the elements generated by the ScalaCheck generator g. We can then use `BatchGen.always[A](bg: Gen[Batch[A]], t: Timeout): Gen[PDStream[A]]`, and `PDStreamGen.always[A](dsg:Gen[PDStream[A]],t: Timeout): Gen[PDStream[A]]` to build more complex generators, using LTL_{ss} operators. We also include combinator functions `next`, `eventually`, `until`, and `release`, that map to LTL_{ss} operators in the obvious way. These combinators are also available as methods for the classes `BatchGen` and `PDStreamGen`, for defining generators easily with a fluent syntax.

We currently do not include combinators for the consume operator λ_x^o, and generators only cover the propositional version of LTL_{ss} from Riesco and Rodríguez-Hortalá (2016b). Also, just like regular ScalaCheck, we do not include functions for non-temporal operators like disjunction, intersection, or implication, and rely on ScalaCheck's `Gen.oneOf` for implementing the disjunction. There are also other combinators to concatenate two `PDStreamGen` objects, both by concatenating the one `PDStream` after the other –combinator `++`– and by concatenating the `PDStream` objects batch by batch, in a zip operation–combinator `+`. Example 1 shows the usage of these combinators.

Test subject `transformation: (DStream[In]) => DStream[Out].` This function that transforms an input DStream into an output DStream is the part of the production code that we are testing with the property.

Assertion `formula: (RDD[In], RDD[Out]).` While the generator defines how to build input DStream prefixes, the assertion formula defines the expected relation between the input DStream and the output DStream. It is a value of type `Formula[(RDD[In], RDD[Out])]`, for `Formula` a sealed trait that is extended by a case class for each operator in LTL_{ss}, following the typical implementation of an algebraic data type in Scala.

https://doi.org/10.1017/S1471068419000012 Published online by Cambridge University Press

This hierarchy includes the case classes `Not`, `Or`, `And`, `Implies`, `Next`, `Eventually`, `Always`, `Until`, and `Release`, that map to LTL_{ss} operators in the obvious way. The consume operator is represented by `case class BindNext[T](timedAtomsConsumer: TimedAtomsConsumer[T])`, where the class `TimedAtomsConsumer[T]` just adds a bit on functionality on top of a given `timedLetterConsumer: Time => T => Formula[T]`, which is a function that defines how to consume the current letter to produce a new formula for the following letter. In this context, `T` would be equal to `(RDD[In], RDD[Out])`, containing the value of the input and output microbatches for the current instant, as corresponds to the Spark interpretation structure $\mathcal{A}^{\mathcal{S}}$. Also, `case class Solved[T](status : Prop.Status)` represents a solved formula with a value in $\{\top, \bot, ?\}$ as correspond to the `status` value. `Prop.Status` is a type defined in the ScalaCheck library, that also includes the undefined value, and that we use to connect sscheck with ScalaCheck. Similarly, to what we did for `PDStream`, the `Formula` trait and its companion object contain functions and methods `or`, `always`, etc, that define a combinator library for formulas.

Regarding other basic formulas, like predicates and equalities, we can represent them as instances of `BindNext`, using constant timed atoms consumer functions. Note, there is no problem with checking these formulas in the next instant, because timeless formulas have the same truth value at all instants. The combinator library offers a couple of ways to express those applications of `BindNext` in a nicer way that direct constructor applications, which in particular integrates with the Specs2 matcher assertions that programmers are familiar with. Examples can be found in Appendix D in the supplementary material.

- The first one is based on the function `Formula.at[T, A, R <% Result](proj: (T) => A)(assertion: A => R): Formula[T]`, which builds a formula by composing a projection function on the current letter, for example, to extract the input batch, with a function that builds a Spec2 matcher with the result—using Specs2's type `Result`—, and uses that to build a `BindNext` instance with a timed atom consumer function that ignores the time argument. In this setting, matchers represent predicates and equalities, and regular Scala functions and methods represent LTL_{ss} functions, which again corresponds to the Spark interpretation structure $\mathcal{A}^{\mathcal{S}}$.
- Another option is directly using the function `Formula.now[T](letterToResult: T => Result): BindNext[T]` that `Formula.at` uses under the hood. There is also `Formula.nowTime[T](letterToResult: (T, Time) => Result): BindNext[T]` for formulas with a time component.
- We also have `Formula.next[T](letterToFormula: T => Formula[T]): Formula[T]` that again builds a `BindNext` instance with nicer syntax, which is called "next" instead of "now" because we do not know if the result formula will be timeless, and so the result in the next instant would not necessarily be applicable to the current instant. There is a version `Formula.nextTime` for using time. In Example 3, we see example usages of `nowTime` and `nextTime`.
- Finally, functions and methods `or`, `always`, etc have variants that accept timed atoms consumer functions, to save some "now" and "next" applications and write the formula more succinctly.

https://doi.org/10.1017/S1471068419000012 Published online by Cambridge University Press

Pretty printer witness `pp1`. This is just an artifact required by ScalaCheck for printing each generated test case while reporting property evaluation results. ScalaCheck already includes implicit values for most usual types, so passing an explicitly value for this argument is rarely required.

Once the generator and the formula are defined, all that is left is using them in a test class that extends Specs2's `Specification` and sscheck's `DStreamTLProperty`. A Specs2 test example can be defined using `DStreamTLProperty.forAllDStream` invoked as `forAllDStream(gen)(transformation)(formula)`, which returns an object of ScalaCheck's type `Prop`, that can be used to launch the property check.

3.3 Verifying AMP Camp's Twitter tutorial

In this section, we give a flavor of the performance of sscheck on a more complex example, adapted for Berkeley's AMP Camp training on Spark,[4] adding sscheck properties for the functions implemented in that tutorial. The code for these examples is reproduced in Appendix D in the supplementary material and it is available for download at Riesco and Rodríguez-Hortalá (2017a), while detailed explanations are available in Riesco and Rodríguez-Hortalá (2018).

Our test subject will work on a stream of *tweets*. A tweet is a piece of text of up to 140 characters, together with some meta-information like an identifier for the author or the creation date. Those words in a tweet that start with the `#` character are called "hashtags" and are used by the tweet author to label the tweet, so other users that later search for tweets with a particular hashtag can locate those related tweets easily. If many tweets use the same hashtag it becomes "popular" (a so-called *trending topic*) and can become a topic of discussion between users. For this reason, Tweeter provides the most popular trending topics in real time, so it is worth noting that popularity is not measured in absolute terms but in a temporal window (i.e., it is more popular a hashtag that appeared 10 times the last minute than one that appeared 20 times yesterday). In these examples, we check the AMP Camp versions of the functions required to compute trending topics. The generators required for checking these functions consist of a stream of random tweets containing hashtags from a certain set given as parameter; we will specify, for each property, the details of the generator. We check the following properties, each corresponding to a Specs2 (Torreborre 2014) example test function.

Hashtags correctly extracted (`getHashtagsOk`). We first check whether the hashtags are correctly computed. We use a simple generator that *always* generates tweets with hashtags in a predefined set. Since we know beforehand, the hashtags in the tweets we check that *always* all the tweets have at least one hashtag, and the computed hashtags are in the set of hashtags indicated by the argument given to the generator.

Hashtags correctly counted (`countHashtagsOk`). We also need to make sure that, for a given period (which does not refer to real but logical time, measured by the number of batches) our functions count all appearances of hashtags. In this case,

[4] http://ampcamp.berkeley.edu/3/exercises/realtime-processing-with-spark-streaming.html

https://doi.org/10.1017/S1471068419000012 Published online by Cambridge University Press

our generator first generates only one hashtag (`#spark`) and after some time only another hashtag (`#scala`), both of them generated with *always*. For this stream, we check that (i) we reach the expected amount of `#spark`, (ii) the amount decreases in the given window *until* it reaches 0, and (iii) we reach the expected amount of `#scala`.

Trending topic correctly found (`sparkTopUntilScalaTop`). We check now that our functions select as trending topic the hashtag that has appeared most often and that this trending topic is updated if another one becomes more popular. We use an *until* generator that produces first tweets with the hashtags `#scalacheck` and `#spark`, using more than twice the latter, and then it keeps generating tweets with `#scalacheck` and `#scala`, but in this case `#scala` appears more than three times for each `#scalacheck`. The corresponding property checks, also with *until*, that the trending topic is correctly computed.

There is always exactly one top hashtag (`alwaysOnlyOneTopHashtag`). Next, we check that there is always just one top hashtag by generating a stream of random tweets and then checking that the output stream of top hashtags always has a single element at each instant in time. This is a simple case of a safety property of the form $\Box result$.

The count of all hashtags is eventually zero (`alwaysEventuallyZeroCount`). We check that the count for all hashtags reaches eventually zero by using a generator that creates a stream of tweets that starts with different hashtags and finishes with tweets without hashtags; this process is repeated inside an *always* operator. Then, we check that it is always the case that eventually the count for all hashtags reaches zero. Note that this property has the form of a liveness property $\Box(\Diamond result)$.

Periodic trending topic (`alwaysPeakImpliesEventuallyTop`). We generate for a long-period random hashtags and a particular hashtag, making sure the latter happens often. Then we check that, if we reach a peak (e.g., 20 appearances in the window) then it corresponds to the particular hashtag we are generating. Note that this property has the form of a liveness property $\Box(premise \rightarrow \Diamond result)$.

In Table 1, we present the execution times for these properties, as well as the number of successful generated tests cases used when checking each property. The test suite was executed on an Intel Core i7-3517U dual core 1.9 GHz and 8 GB RAM, with Spark running in local mode. That is a reasonable time for an integration test and could be used as an automated validation step in a continuous integration pipeline (Fowler and Foemmel 2006). sscheck local execution could be also used for local developing to fix a broken test, using a longer batch interval configuration and smaller number of passing test cases to adapt to a scenario with less computing resources. On the other hand, if a cluster is available, sscheck could be executed using Spark distributed mode – by setting the `sparkMaster` field appropriately – using a shorter batch interval, higher default parallelism, and a higher number of passing tests. In the future, we also plan to develop a new feature to allow several test cases for the same property to execute in parallel. This is not trivial because Spark is limited to a single Spark context per Java virtual machine (JVM) – see https://issues.apache.org/jira/browse/SPARK-2243.

https://doi.org/10.1017/S1471068419000012 Published online by Cambridge University Press

Table 1. *AMP Camp's sscheck properties*

Property	Exec. time (s)	Test cases
`getHashtagsOk`	46	10
`countHashtagsOk`	142	15
`sparkTopUntilScalaTop`	56	15
`alwaysOnlyOneTopHashtag`	45	10
`alwaysEventuallyZeroCount`	187	15
`alwaysPeakImpliesEventuallyTop`	303	15

4 Related work

At first sight, the system presented in this paper can be considered as an evolution of the data-flow approaches for the verification of reactive systems developed in the past decades, exemplified by systems like Lustre (Halbwachs 1992) and Lutin (Raymond et al. 2008). In fact, the idea underlying both SPS and data-flow reactive systems is very similar: processing 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 supporting more expressive formulas would be an interesting subject of future work, in this work, we have focused on 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 SPS 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.

https://doi.org/10.1017/S1471068419000012 Published online by Cambridge University Press

In a similar note, we can consider runtime monitoring of synchronous systems like Lola (D'Angelo et al. 2005), 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, as 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 (Barringer and Havelund 2011) is a Scala library that implements a logic for analyzing sequences of events (traces). 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 supports a form of first-order quantification over events. The logic is implemented as a shallow internal DSL, just as we do for LTL_{ss} in sscheck, and it also supports stepwise evaluation of traces so it can be used for online monitoring of a running system, besides evaluating recorded execution 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 (Karau 2015a), 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 future work

In this paper, we have presented sscheck, a property-based testing tool for Spark Streaming programs. sscheck allows the user to define generators and to state properties using LTL_{ss}, an extension of LTL with timeouts in temporal operators and a special operator for binding the current batch and time. This logic allows us to define a stepwise transformation that only requires/generates the current batch; using this feature, the Scala implementation of sscheck takes advantage of lazy functions to efficiently implement the tool. The benchmarks presented in the paper show that the approach works well in practice. With these features in mind, we hope sscheck will be accepted by the industry; we consider the presentation at Apache Europe (Riesco and Rodríguez-Hortalá 2016a) and citations in books written by remarkable members of the Spark community (Karau and Warren 2017) are important steps in this direction.

There are many open lines of future work. First, adding support for arbitrary nesting of ScalaCheck `forall` and `exists` quantifiers inside LTL_{ss} formula would be an interesting extension. Moreover, we also consider developing versions for other languages with Spark API, in particular Python, or supporting other SPS, like Apache Flink (Carbone et al. 2015b) or Apache Bean (Akidau et al. 2015). This would require novel solutions, as these systems are not based on synchronous microbatching but they process events one at a time, and also have interesting additional features like the capability for handling different event time characteristics for supporting out-of-order streams and several types of windows (Akidau et al. 2015). Besides, we plan to explore whether the execution of several test cases in parallel minimize the test suite execution time. We could also improve the sscheck library interface, employing advanced Scala DSL techniques like the Magnet Pattern (Typesafe Inc. 2012) to make formulas easier to write and read. Finally, we intend to explore other formalisms for expressing temporal and cyclic behaviors (Wolper 1983).

https://doi.org/10.1017/S1471068419000012 Published online by Cambridge University Press

Supplementary material

To view supplementary material for this article, please visit https://doi.org/10.1017/S1471068419000012.

References

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

Akidau, T., Bradshaw, R., Chambers, C., Chernyak, S., Fernández-Moctezuma, R. J., Lax, R., McVeety, S., Mills, D., Perry, F., Schmidt, E., et al. 2015. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment 8*, 12, 1792–1803.

Alur, R. and Henzinger, T. A. 1994. A really temporal logic. *Journal of the ACM 41*, 1, 181–204.

Barringer, H. and Havelund, K. 2011. Tracecontract: A scala DSL for trace analysis. In *Proceedings of the 17th International Symposium on Formal Methods, FM 2011*, vol. 6664, "emLecture Notes in Computer Science, M. J. Butler and W. Schulte, Eds. Springer, Berlin Heidelberg, 57–72.

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

Beck, K. 2003. *Test-Driven Development: By Example*. Addison-Wesley Professional, Boston, USA.

Blackburn, P., van Benthem, J. and Wolter, F., Eds. 2006. *Handbook of Modal Logic*. Elsevier, Amsterdam, the Netherlands.

Carbone, P., Ewen, S., Haridi, S., Katsifodimos, A., Markl, V. and Tzoumas, K. 2015a. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering 38*, 4, 11.

Carbone, P., Fóra, G., Ewen, S., Haridi, S. and Tzoumas, K. 2015b. Lightweight asynchronous snapshots for distributed dataflows. *arXiv preprint, arXiv:1506.08603*.

Claessen, K. and Hughes, J. 2011. QuickCheck: A lightweight tool for random testing of Haskell programs. *Acm Sigplan Notices 46*, 4, 53–64.

D'Angelo, B., Sankaranarayanan, S., Sánchez, C., Robinson, W., Finkbeiner, B., Sipma, H. B., Mehrotra, S. and Manna, Z. 2005. LOLA: Runtime monitoring of synchronous systems. In *Proceedings of the 12th International Symposium on Temporal Representation and Reasoning, TIME 2005*. IEEE Computer Society, 166–174.

Fitting, M. and Mendelsohn, R. L. 1998. *First-Order Modal Logic*, vol. 277, *Science & Business Media*. Springer, Berlin Heidelberg.

Fowler, M. and Foemmel, M. 2006. Continuous integration. *Thought-Works*, Addison-Wesley, Boston, USA, 122.

GoogleTrends. 2018. Google trends for spark streaming. trends.google.com. https://trends.google.com/trends/explore?date=2012-01-01%202018-03-01&q=spark%20streaming,apache%20storm,apache%20flink,akka%20streams,apache%20beam.

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

Halbwachs, N. 1992. *Synchronous programming of reactive systems*, vol. 215, *Springer International Series in Engineering and Computer Science*. Kluwer Academic Publishers, Berlin Heidelberg.

https://doi.org/10.1017/S1471068419000012 Published online by Cambridge University Press

Karau, H. 2015a. Spark-testing-base. http://blog.cloudera.com/blog/2015/09/making-apache-spark-testing-easy-with-spark-testing-base/.

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

Karau, H. 2015c. Effective testing of spark programs and jobs. In *Strata + Hadoop World 2015 NYC*. O'Reilly. https://conferences.oreilly.com/strata/big-data-conference-ny-2015/public/schedule/detail/42993.

Karau, H. and Warren, R. 2017. *High Performance Spark: Best Practices for Scaling and Optimizing Apache Spark*. O'Reilly Media, Incorporated, Missouri, USA.

Kuhn, R. and Allen, J. 2014. Reactive Design Patterns. Manning Publications Co. 2017, New York, USA.

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

Marz, N. and Warren, J. 2015. *Big Data: Principles and best practices of scalable realtime data systems*. Manning Publications Co, New York, USA.

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

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

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

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

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

Raymond, P., Roux, Y. and Jahier, E. 2008. Lutin: A language for specifying and executing reactive scenarios. *EURASIP Journal on Embedded Systems 2008*, 753821.

Riesco, A. and Rodríguez-Hortalá, J. 2015–2017a. Examples using sscheck. https://github.com/juanrh/sscheck-examples.

Riesco, A. and Rodríguez-Hortalá, J. 2015–2017b. sscheck: Scalacheck for spark v0.3.2. https://github.com/juanrh/sscheck/releases/tag/0.3.2. See ScalaDoc documentation at https://juanrh.github.io/doc/sscheck/scala-2.10/api, and basic setup instructions at https://github.com/juanrh/sscheck/wiki/Quickstart.

Riesco, A. and Rodríguez-Hortalá, J. 2016a. Property-based testing for Spark Streaming. In *Apache Big Data Europe 2016*. The Linux Foundation. http://events.linuxfoundation.org/events/apache-big-data-europe/program/schedule.

Riesco, A. and Rodríguez-Hortalá, J. 2016b. Temporal random testing for spark streaming. In *Proceedings of the 12th International Conference on integrated Formal Methods, iFM 2016*, vol. 9681, *Lecture Notes in Computer Science, E. Abraham and M. Huisman, Eds.* Springer.

Riesco, A. and Rodríguez-Hortalá, J. 2018. Property-based testing for spark streaming (extended version). Technical Report 02/2018, Departamento de Sistemas Informáticos y Computación de la Universidad Complutense de Madrid, Berlin Heidelberg. http://maude.sip.ucm.es/~adrian/pubs.html.

Smullyan, R. M. 1995. *First-Order Logic*. Courier Corporation.

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

Typesafe Inc. 2012. The magnet pattern. http://spray.io/blog/2012-12-13-the-magnet-pattern/.

https://doi.org/10.1017/S1471068419000012 Published online by Cambridge University Press

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

White, T. 2012. *Hadoop: The Definitive Guide*. O'Reilly Media, Missouri, USA.

Wolper, P. 1983. Temporal logic can be more expressive. *Information and Control 56*, 1/2, 72–99.

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

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

https://doi.org/10.1017/S1471068419000012 Published online by Cambridge University Press