Received September 17, 2019, accepted October 10, 2019, date of publication October 14, 2019, date of current version October 28, 2019.

*Digital Object Identifier 10.1109/ACCESS.2019.2947361*

# FlinkCheck: Property-Based Testing for Apache Flink

**CRISTINA VALENTINA ESPINOSA[1], ENRIQUE MARTIN-MARTIN[2], ADRIÁN RIESCO[2], AND JUAN RODRÍGUEZ-HORTALÁ[3]**
[1]Independent Researcher
[2]Departamento de Sistemas Informáticos y Computación, Facultad de Informática, Universidad Complutense de Madrid, 28040 Madrid, Spain
[3]Departamento de Ingeniería de Software e Inteligencia Artificial, Universidad Complutense de Madrid, 28040 Madrid, Spain

Corresponding author: Adrián Riesco (ariesco@ucm.es)

This work was supported in part by the Santander-Universidad Complutense de Madrid (UCM) Project under Grant PR26/16-21B-2, in part by the Comunidad de Madrid projects BLOQUES-CM under Grant S2018/TCS-4339 and FORTE-CM under Grant S2018/TCS-4314, co-funded by the Fondos Estructurales y de Inversión Europeos (EIE) of the European Union, in part by the Ministerio de Economía y Competitividad (MINECO) projects TRACES under Grant TIN2015-67522-C3-3-R and LOBASS under Grant TIN2015-69175-C4-2-R, and in part by the Spanish Ministerio de Ciencia e Innovación/Fondo Europeo de Desarrollo Regional (MICINN/FEDER), UE project, under Grant RTI2018-094403-B-C31.

**ABSTRACT** Apache Flink is an open-source, soft real-time stream processing framework underlying many modern systems dealing with cloud and real-time computing, data analytics, and the Internet of Things, among others. As the complexity of stream-processing systems increase, the testing, debugging, and verification tools supporting them should improve as well. However, Flink's testing tools only include a local cluster fake, which requires a great effort from the user to manually craft all those streams (and their corresponding output) that are relevant for each particular function under test. Property-based testing is an automatic, black-box testing technique that tests functions by generating random inputs and checking whether the obtained outputs fulfill a given property. In this paper, we present FlinkCheck, a property-based testing tool for Apache Flink. It uses a bounded temporal logic for both guiding how random streams are generated and defining the properties. We illustrate how the tool works with an example of a collaborative initiative against sexual harassment.

**INDEX TERMS** Apache Flink, stream processing, property-based testing, linear temporal logic (LTL).

## I. INTRODUCTION

Apache Flink [1] is a soft real-time stream processing framework supporting stateful computations over unbounded and bounded data streams. Flink can be executed in a distributed way in several popular cluster environments, and provides good performance through features like operator pipelining, efficient type aware serialization, and data locality. These features make Flink suitable for a wide range of modern systems requiring efficient soft real-time computation, such as Amazon Kinesis Data Analytics [2], Alibaba Realtime Compute [3], and Huawei Cloud Stream [4].

Stream processing is used in sophisticated use cases like data analytics and anomaly detection, that lead to complex systems where appropriate quality assurance procedures are required. The strategy for testing streaming programs described in Flink's documentation [5] suggests using classical unit testing frameworks [6] for testing individual operators, and a local cluster fake [6] to test a full program, and also references some internal testing tooling with unstable interfaces for testing checkpoints and state handling. However, in all those cases the user is required to introduce by hand the input stream and the expected output, which might be huge in general, making these testing techniques tedious and error-prone. Moreover, the documentation itself indicates that state handling testing "could be tricky because of time dependencies" [5], which emphasizes the need for stream-oriented testing techniques.

In this paper, we present FlinkCheck, a property-based testing (PBT) tool for Apache Flink. FlinkCheck provides a bounded temporal logic for generating inputs for functions and for stating properties. This logic, devised for streaming systems in [7], allows users to define how streams vary with time and what properties should verify the corresponding output. FlinkCheck randomly generates a specified number of finite input stream prefixes with the required structure and evaluates the output streams produced by the Flink runtime. This allows for using longer stream prefixes that exercise

The associate editor coordinating the review of this manuscript and approving it for publication was Ahmed M. Elmisery.

the test subject more thoroughly, leaving the user only the creative tasks. Empirical studies [8], [9] have shown that PBT techniques obtain good code coverage results in practice, with smaller effort compared to manually writing tests that cover all cases using xUnit frameworks. Besides this general contribution, the presentation of the property-based testing tool FlinkCheck, the specific contributions of the paper are:

- The instantiation of the logic for Apache Flink.
- The translation from mathematical notation to Flink.
- An example illustrating how to use FlinkCheck and how to understand the results.
- Benchmarks to check the practicability and scalability of FlinckCheck.

The rest of the paper is structured as follows: Sect. II briefly describes the foundations of FlinkCheck. Sect. III presents the design of the tool, while Sect. IV describes how to use it and its performance with some benchmarks. Sect. V discusses the related work. Finally, Sect. VI concludes and outlines some lines of future work. The source code of FlinkCheck, examples, and more information is available at https://github.com/demiourgoi/flink-check.

## II. PRELIMINARIES

In this section, we present the main ideas underlying our system: the Flink foundations, property-based testing, and linear temporal logic.

### A. FLINK

Flink is a distributed processing engine that supports both stateful stream processing and batch processing, by treating the later as a particular case of the former where the input streams are finite—*bounded* in Flink's terminology [10]. Flink's main abstraction for stream processing is the DataStream type, which corresponds to a potentially infinite—unbounded—list of elements of a given type. Streams are immutable, but derived streams are defined by transforming an existing stream using operators inspired by functional programming, such as map and filter higher-order functions, or Hadoop [11] influenced operators using key-value pairs, that allow for splitting streams into partitions that are materialized independently in different worker processes running on a cluster. In order to incrementally produce an output while processing an unbounded stream, the programmer can use Flink's API to define a policy to split the stream into *windows* of finite size, that are aggregated to produce intermediate results [12]. Windows can be defined in terms of time (e.g. a *tumbling window* every 3 minutes, or a *sliding window* of 3 minutes each minute), in terms of elements (e.g. a window every 20 received elements), and based on sessions (related events separated by less than a configured gap of time).

Another remarkable feature of Flink is that it supports different notions of time [12], namely processing, event, and ingestion time. With *event time*, the timestamp for each element of a stream is computed from the event using a specified function, usually by selecting an element field. As a result, the stream is dynamically reordered, which is useful for situations where there are nondeterministic delays between event generation and event ingestion, e.g. when sending data from a set of IoT resources in different locations with limited networking capabilities. With *processing time*, when a Flink operator processes a stream element, it uses the current system clock time of the machine where it is running as the element's timestamp. With *ingestion time* elements are assigned a processing time timestamp in the stream source operator, that is kept for the remaining operators. Element timestamps then determine how time-based windows are constructed. Event time leads to more predictable and deterministic programs and facilitates certain use cases, but it is more costly because stream reordering is performed by buffering events. Processing time is less expressive but also less costly, and ingestion time lays in between.

In this paper, we focus on using Flink for stream processing, and in writing tests using Flink's Scala API, which allows us to use the PBT tool ScalaCheck [13] as the basis for our prototype. Scala [14] is a programming language that combines object-oriented features and functional programming, and it is a good fit due to Flink's functional programming influences.

```
case class Incident(zone_id: Int, danger: Double)

object DangerLevel extends Enumeration {
  type DangerLevel = Value
  val Safe, Warning, Danger, Extreme = Value

  def apply(danger: Double) = danger match {
    case x if x <= 1.0 => Safe
    case x if (x > 1.0) && (x <= 5.0) => Warning
    case x if (x > 5.0) && (x <= 8.0) => Danger
    case _ => Extreme
  }
}

def harass_max(raw: DataStream[Incident]):
    DataStream[(Int, DangerLevel)] =
  raw.keyBy("zone_id")
     .timeWindow(Time.hours(1))
     .max("danger")
     .map {x => (x.zone_id,
                 DangerLevel(x.danger))}
```

**FIGURE 1.** HarassMap Flink program.

We illustrate these ideas with a simple collaborative example inspired by *HarassMap* [15], [16], whose code can be found in Fig. 1. HarassMap is an Egyptian mobile and web application that collects information about sexual harassment incidents from witnesses or victims to create a map showing harassment hot spots. HarassMap manually verifies the incidents and volunteers visit those neighborhoods to raise awareness of sexual harassment behaviors. We will assume that our Flink program receives an unbounded stream of verified harassment incidents generated by people. Each incident is represented as Incident objects, which contain a zone identifier zone_id (natural number) and a danger value (real number between 0 and 10)—see the case class Incident in Fig. 1. We aim to label each zone with a level depending on the maximum perceived danger in a time window. Concretely, we will consider that a zone with a maximum danger up to 1 is Safe, between 1 and 5 the level is Warning, between 5 and 8 the level is Danger, and above 8 is Extreme. These levels are represented in Scala using an enumeration type DangerLevel whose apply method translates values to levels. Finally, our Flink program is implemented in the function harass_max of Fig. 1. It processes an input stream of incidents, with type DataFrame[Incident], and creates an output stream of pairs DataStream[(Int, DangerLevel)] where the first component is the zone identifier and the second one is the danger level corresponding to the maximum perceived danger in intervals of one hour: The function harass_max first groups the incoming incidents by their zone identifier (transformation keyBy) and creates one-hour windows (transformation timeWindow) to process together all the incidents within one hour. Then, it computes the maximum value for each zone in the time window (using max) and transforms this value into the appropriate danger level by using the Flink DataStream transformation map. This transformation accepts an anonymous function that given an Incident object generates a pair with the zone_id and the level corresponding to the danger value. The complete code can be found in the file Harass.scala at https://git.io/fji9K.

## B. PROPERTY-BASED TESTING

Property-based testing (PBT) [9], [13], [17]–[19] is a black-box, automatic testing technique that consists of two ingredients: (a) *generators*, which produce random values for the input parameters of the test subject function, and (b) *formulas*, which are Boolean expressions that define a predicate that relates the input and the output of a test subject function. We combine generators and assertions under a universal quantifier to form a *property*, that is a first-order logic formula about the test subject that is checked during testing.

Our tool extends ScalaCheck [13], a Scala PBT library that provides generators for built-in Scala types. Using ScalaCheck we could define a simple property for the list reverse function as:

```
forAll (listOfN(10, posNum[Int]))
       { list => list.reverse.reverse == list }
```

where the forAll property first receives a generator (listOfN) that produces lists of 10 positive numbers (produced in turn with the built-in ScalaCheck posNum generator). Then, the formula requires that applying reverse twice to any list returns the same list.

For Flink streaming programs this kind of generators/properties is not enough because they do not take time into account. For example, in the example in the previous section, we would need to create finite streams and check the corresponding outputs. We would define first the generators we want to use (e.g. a data stream with all windows containing incidents with a danger value greater than 1, or a data stream that alternates values corresponding to different danger levels) and then state the properties the corresponding output streams should fulfill (e.g. never Safe is returned or all danger levels are generated). In the next section, we present a logic to extend property-based testing to take time into account.

## C. LINEAR TEMPORAL LOGIC FOR STREAM PROCESSING SYSTEMS

FlinkCheck uses the $LTL_{ss}$ logic [7], an extension of propositional linear temporal logic (LTL) [20] with timeouts in temporal operators, and atomic formulas constructed with predicates applied to terms. $LTL_{ss}$ is not first-order because it just includes a simple variable binding construct instead of the usual quantifiers. Judgments in this logic evaluate a formula and a finite *word* that uses terms paired with a timestamp as *letters*, to a three-valued truth value $v \in \{\top, \bot, ?\}$, where ? corresponds to inconclusive cases where the word is too short to determine whether the formula holds or not. It is important to note that, although the underlying logic is the same, the time model used in FlinkCheck makes the corresponding instantiation of $LTL_{ss}$ completely independent of previous approaches.

Next, we present generators and formulas for $LTL_{ss}$. Although the logic is general and hence it can be adapted to any stream processing system, our presentation will directly focus on Flink concepts and consider letters as windows and words as streams. We will also present examples based on the *HarassMap* example in the previous section to illustrate the intuitive ideas. Then, in Sect. III we will see how FlinkCheck implements these elements.

### 1) GENERATORS

We explain in this section how to generate values with $LTL_{ss}$, using $g_1$ and $g_2$ as general generators. Generators are defined as:

1) Basic generators correspond to regular ScalaCheck generators. For the example above, we could define a generator for random numbers, pairs of random numbers, and lists of pairs of random numbers. This list of pairs would correspond to a Flink window, so time is not involved in basic generators. In the rest of the section, we assume we have defined the basic generators $g_s$, $g_w$, and $g_d$, which generate lists of pairs $(id, n)$ with $id$ a natural number and $n \leq 1$, $1 < n \leq 5$, and $5 < n \leq 8$, respectively.
2) $X\ g_1$ (next $g_1$), which generates an empty window and uses $g_1$ for the second one. For example, the generator $X\ g_s$ would generate a stream with two windows, the first one empty and the second one containing pairs of numbers with the second component smaller than or equal to 1.
3) $\Box_n\ g_1$ (always $g_1$ for $n$), which generates $n$ consecutive windows using $g_1$. For example, $\Box_4\ g_w$ generates a

stream with 4 windows. Each window will contain pairs of numbers with the second component of the pair a number $n$ with $1 < n \leq 5$.

4) $\Diamond_n\ g_1$ (eventually $g_1$ for $n$), which generates between 1 and $n-1$ empty windows and then a final window using $g_1$. For example, $\Diamond_4\ g_d$ generates a stream with *at most* 4 windows; the last window will contain pairs of numbers with the second component of the pair a number $n$ with $5 < n \leq 8$.
5) $g_1\ U_n\ g_2$ ($g_1$ until $g_2$ for $n$), which generates values using $g_1$ for less than $n$ windows and then uses $g_2$ once. For example, $g_d\ U_6\ g_s$ might generate, among others, a stream of 4 windows; all windows contain pairs of numbers $(id, n)$, but for the first three windows we have $5 < n \leq 8$ and for the last one $n \leq 1$.
6) Finally, FlinkCheck provides composition operators for generators. For example, $+$ can be used to combine two basic generators to obtain a generator that produces the union of the two windows generated by its arguments. $+$ is also overloaded for combining general generators, so given $g_1$ and $g_2$ returns a generator $g_1 + g_2$ as the pairwise union. It first generates a pair of streams $(s_1, s_2)$ from $g_1$ and $g_2$, and then returns a stream where each window at position $i$ is the union of the windows at position $i$ in $s_1$ and $s_2$, filling up with empty windows in case $s_1$ is longer than $s_2$, or vice versa.

### 2) FORMULAS

In this section, we present $LTL_{ss}$ formulas, using $f_1$ and $f_2$ as general formulas. Formulas can be evaluated to either true ($\top$), false ($\bot$), or undefined result (?) if the given stream does not contain enough windows to evaluate it. This contrasts with generators, that always generate a stream that would make the formula corresponding to the generator evaluate to true. Formulas are defined as:

1) Basic formulas are Boolean functions and constants. In the rest of the section, we assume a function $danger(s)$ that holds if there is a pair $(id, n)$ in the set $s$ such that $n$ is greater than 5, while $warning(s)$ holds if the pair $(id, n)$ with the highest value in $s$ for $n$ satisfies $1 < n \leq 5$. Note that time is not involved in basic formulas.
2) $\lambda^{t}_{(i,o)} f_1$, read *consume i, o, and t to evaluate* $f_1$. This expression evaluates the formula $f_1$ after binding the corresponding variables: $i$ to the current input window, $o$ to the current output window, and $t$ to the current timestamp. In general, we will omit the timestamp when not required. This operator allows us to apply Boolean functions to the streams so, combined with the temporal operators below, provides the full capacity of the logic.
3) $X\ f_1$, which holds if the formula $f_1$ holds for the next window.
4) $\Box_n\ f_1$, which holds if $f_1$ holds for all of the first $n$ windows. For example, the formula $\Box_4 \lambda_{(i,o)} warning(i)$ (a) holds for the stream $\Box_4\ g_w$; (b) fails for the stream $\Box_4\ g_s$; and (c) returns ? for the stream $\Box_3\ g_w$ (although it holds for the first three states the tool cannot tell what will happen in the fourth one).
5) $\Diamond_n\ f_1$ holds if $f_1$ holds for any of the first $n$ windows. For example, the formula $\Diamond_4 \lambda_{(i,o)} warning(i)$ (a) holds for the streams $\Box_4\ g_w$ and $\Box_3\ g_w$; (b) fails for the stream $\Box_4\ g_s$; and (c) returns ? for the stream $\Box_3\ g_s$.
6) $f_1\ U_n\ f_2$ holds if $f_1$ holds for the first $k$ windows ($0 \leq k < n$) and $f_2$ holds for the $k+1$ window. For example, $(\lambda_{(i,o)} warning(i))\ U_4\ (\lambda_{(i',o')} danger(i'))$ (a) holds for the streams $\Box_2\ g_d$ ($f_1$ holds for 0 windows and $f_2$ holds in the initial window) and $\Box_2\ g_w + X\ X\ g_d$ ($f_1$ stops holding in the third window but then $f_2$ holds); (b) fails for the stream $\Box_2\ g_w + X\ X\ g_s$ (in the third window we reach a state where $f_1$ stops holding but $f_2$ does not hold yet); and (c) returns ? for the stream $\Box_3\ g_w$ ($f_1$ keeps holding, so it might still be the case that $f_2$ will hold in the next window). Note how the logical operator $+$ is used in combination with two applications of the next operator ($X$) to require that, in the third window, a particular set of elements is required; meanwhile, the first two windows are populated by the $\Box_2\ g_w$ generator.

## III. FLINKCHECK DESIGN

In this section, we describe the implementation of FlinkCheck, how it differs from our previous work on testing Spark Streaming [7], and how the peculiarities and characteristic features of Flink motivate those changes. Note that the different time models used by these tools make the corresponding instantiations different, although some low level libraries are shared thanks to a careful design of the whole project available at https://github.com/demiourgoi.

### A. DESIGN OVERVIEW

FlinkCheck's code is structured in two libraries: *sscheck-core* and *flink-check*. *sscheck-core* is shared with Sscheck [7] and implements $LTL_{ss}$ as a set of ScalaCheck generators and as a Scala Formula trait for specifying properties. *sscheck-core* is agnostic from Flink or any particular computing engine. As $LTL_{ss}$ is a discrete time logic, in *sscheck-core* generators produce streams represented as sequences of windows, where each window is a multiset of elements that occur during the time interval corresponding to the window —similarly to the discrete representation of [21]. Those windows have no associated timestamp, so we can interpret them using different windowing policies for different computing engines. On the other hand, Formula is a generic type parameterized by a type for the letters, so properties see streams as sequences of letters with an associated timestamp.

The *flink-check* library builds on top of *sscheck-core*, and allows us to define ScalaCheck properties for Flink by extending the trait DataStreamTLProperty, which includes the `forAllDataStream` method shown in Fig. 2.[1] This method

[1]Implicit parameters have been omitted for the sake of conciseness.

```
type TSGen[A] = Gen[Seq[TimedElement[A]]]
type Letter[In, Out] = (DataSet[TimedElement[In]],
      DataSet[TimedElement[Out]])

def forAllDataStream(generator: TSGen[In])(
   subject: (DataStream[In]) => DataStream[Out])(
  formula: FlinkFormula[Letter[In, Out]]): Prop
```

**FIGURE 2.** **forAllDataStream method.**

returns a ScalaCheck property that checks that for all test cases obtained from the generator, the property holds when using the test case as the input stream and the result of applying the test subject as the output stream. That wraps the propositional temporal formula formula —as $LTL_{ss}$ is a propositional logic— with a single universal quantifier. As usual in ScalaCheck and PBT in general, the test will try to refute this formula by generating a specified number of test cases and checking formula for them, but no sophisticated automated reasoning procedure is used to evaluate the property. Here we use

case class TimedElement[T](timestamp: Long, value: T)

to represent elements together with the time they occur in the scenario simulated by the test case. Note that test subjects are Flink stream transformations that can be defined using any of the operations in the Flink API. Each generated test case is executed in two stages. First, during the *test case exercise* phase, the test case is transformed from a discrete sequence of windows into a continuous Flink DataStream (see Sect III-B), which is fed to the test subject to get an output stream. The input and output streams are then stored by Flink in its configured default file system. Later, during the *test case evaluation* phase, the input and output streams are read from the file system, and discretized using the specified windowing policy (see Sect III-C), obtaining a sequence of pairs of Flink DataSet corresponding to the timewise pairing of the window sequences for the input and output streams. That sequence is fed into the property formula, that uses those windows pairs as letters, and checks whether the expected relation between input and output streams holds.

Those conversions between continuous streams and discrete window sequences allow us to use the discrete $LTL_{ss}$ logic, but we think that is also natural for Flink, where splitting streams into windows and producing intermediate aggregated results for each window is a basic programming idiom [12]. This is the same idea, but evaluating assertions on each pair of windows to incrementally evaluate the property formula. Also, by splitting test case exercise and evaluation we can execute the test subject without interference from the assertions, that can use arbitrary Flink API operations that perform potentially costly computations that could slow down the test subject too much for stream processing applications [22].

The current implementation of FlinkCheck, like other existing testing tools included with Flink [5], only supports execution in a single host. That is enough for testing the functional correctness of Flink programs, by generating small data streams that fit into a single host, but that exercise the logic of the test subject. In fact, a testing tool that runs locally in a single host is convenient for automating the execution of the tests on continuous integration services like e.g. Travis CI [23], AWS CodeBuild,[2] or CircleCI,[3] that only provide a single Docker container for running the tests. As we will see in Sect. IV-C, the current implementation achieves reasonable times for running a suite of functional tests. Besides that, in Sect. VI we will discuss possible future extensions for using FlinkCheck with a cluster, and some use cases enabled by that. Now let us see more details about FlinkCheck's generators and properties.

### B. GENERATORS

The object FlinkGenerators (see https://git.io/fjPrp) in *flink-check* takes care of converting the discrete window sequence generators in *sscheck-core* into generators of Seq[TimedElement[In]]. Its function tumblingTimeWindows does that by interpreting the generated window sequences as tumbling windows [12], assigning to each particular window element a random timestamp uniformly distributed between the start and the end of the window. For now we only support tumbling windows, but the programmer can also directly pass a ScalaCheck generator for Seq[TimedElement[In]] to forAllDataStream, as long as the generated elements appear in increasing order of timestamp, and we plan to support other windowing policies out of the box in the future. The generated sequence is converted into a DataStream[In] by using Flink's built-in function StreamExecutionEnvironment.fromCollection, and then assigning the timestamp to each element and just keeping the values with stream.assignAscendingTimestamps (_.timestamp).map(_.value). This shows how we use Flink's event time as a key ingredient for making FlinkCheck work. We made this choice to make tests repeatable and deterministic, as event time is not dependent on the performance of the hardware where tests are running. Test subjects do not need to be aware of event time, that is configured by default by FlinkCheck, but for test subjects where event time is relevant, the function FlinkGenerators.eventTimeToFieldAssigner can be used to modify the generated elements so their relevant fields are consistent with the generated timestamps. Note this design based on fromCollection currently limits FlinkCheck to generating test cases that fit in the memory of a single host.

In turn, tumblingTimeWindows can be used to adapt any of the generators defined by *sscheck-core*'s WindowGen (see https://git.io/fjPoI), which covers every generator presented in Sect. II-C1, as can be seen in Table 1. Here, the parameters g are generators of *one sequence of elements*, e.g. the contents of a window. As before, WindowGen provides the helper

[2] https://aws.amazon.com/codebuild/
[3] https://circleci.com

**TABLE 1.** $LTL_{ss}$ **generators and their FlinckCheck counterparts.**

| $LTL_{ss}$ generator | FlinkCheck generator |
|---|---|
| $X\ g$ | WindowGen.next(g) |
| $\Box_n\ g$ | WindowGen.always(g,n) |
| $\Diamond_n\ g$ | WindowGen.eventually(g,n) |
| $g_1\ U_n\ g_2$ | WindowGen.until(g1,g2,n) |

functions ofN(n,ge) and ofNtoM(n,m,ge) to easily create one sequence of exactly n elements or between n and m elements generated by ge. Generators ge create *exactly one element*, and can be defined in Scala using the standard ScalaCheck generators in the Gen module. For example, the generator of a DataStream containing 24 windows of 1 hour with 10 random integer numbers between 0 and 1000 each window would be defined as:

```
tumblingTimeWindows(Time.hours(1)){
  WindowGen.always(
    WindowGen.ofN(10,Gen.chooseNum[Int](0,1000)),
    24)
}
```

Finally, *sscheck-core*'s PStreamGen also defines functions next, always…for combining generators of sequences of windows like those returned e.g. by WindowGen.always.

### C. $LTL_{ss}$ FORMULAS

Similarly to generators, *flink-check*'s class FlinkFormula wraps a *sscheck-core* Formula together with a StreamDiscretizer that implements a windowing policy for splitting a DataSet[TimedElement[T]] into a sequence of windows represented by value of the type TimedWindow:

```
case class TimedWindow[T](timestamp: Long,
  data: DataSet[TimedElement[T]])
```

During test exercise, the input and output data streams are stored pairing each element with its timestamp, so when we read the serialized streams we can reconstruct not only the window start timestamp but also the timestamp of each individual element. The stream discretizer aligns with the start of the first window generated by tumblingTimeWindows by using the same optional start time parameter for both (with the same default value). That alignment allows FlinkCheck to use empty windows safely. Currently, discretizers for tumbling windows and sliding windows are available. A different windowing policy can be used for the generator and the property formula, as we will see in Sect IV-B. *sscheck-core*'s Formula (https://git.io/fjP6I) provides functions for every $LTL_{ss}$ formula operator presented in Sect. II-C2, as can be seen in Table 2. Any Boolean expression is promoted to an $LTL_{ss}$ formula with Solved. The formulas *next*, *always*, *eventually*, and *until* are represented with the corresponding functions. The major difference is that the number of windows $n$ is not part of the function but is added by the decorator during. Finally, the *consume* operation is represented by the function consume(f). In its simplest case, the parameter fr is

**TABLE 2.** $LTL_{ss}$ **formulas and their FlinkCheck counterparts.**

| $LTL_{ss}$ formula | FlinkCheck formula |
|---|---|
| Boolean expr. $b$ | Solved(b) |
| $X\ f$ | Formula.next(f) |
| $\Box_n\ f$ | Formula.always(f) during n |
| $\Diamond_n\ f$ | Formula.eventually(f) during n |
| $f_1\ U_n\ f_2$ | f1.until(f2) during n |
| $\lambda^t_{(i,o)} f$ | Formula.consume(fr) |

a function that receives a pair $(i, o)$ and returns a Specs2 [24] Result. The parameter passing of the function fr performs the binding of the current letter, and the Result value is the outcome of an assertion. This enables using Specs2 matchers in the properties, like the additional Flink specific matchers included with FlinkCheck: e.g. foreachElement is a matcher that checks whether all elements in a DataSet fulfill a predicate (see Sect. IV-A later). *consume* can also take a function that returns a new formula, for building complex properties with nested temporal operators.

Once a Formula is defined, the groupBy decorator is used to specify the window policy for the stream discretizer, obtaining a FlinkFormula. For example, always(f) during 3 groupBy TumblingTimeWindows(Time.minutes(15)) expresses that the formula f must be satisfied for the next 3 tumbling windows of 15 minutes. Finally, the Formula object also contains several variants for these functions so that programmers can define the properties more concisely. For example, the eventually function has a version eventuallyF that instead of a formula accepts a function from a pair $(i, o)$ to a formula. This version simplifies formulas like $\Diamond_n(\lambda^t_{(i,o)} f)$ where *consume* is inside *eventually*.

## IV. USING FLINKCHECK

In this section, we describe how to define generators and properties in FlinkCheck, and how to use them to test a function that transforms a Flink stream. In particular, we use the collaborative example in Sect. II-A to present how safety (something incorrect never happens) and liveness (it is always the case, given a premise, that something good eventually happens) properties can be defined. We also show benchmarks using the mentioned properties to show the scalability and practicability of the tool. More complex examples can be found in the *flink-check-examples* subproject (https://git.io/JeOnn), including FlinkCheck properties for the Flink training from Ververica [25], a company founded by the creators of Flink that is devoted to its commercial development.

### A. WRITING A SAFETY PROPERTY

The safety property we want to test states that, given a stream of incidents where all danger values are greater than 1, then the level Safe should never be generated. We first define a "timeless" generator of Incident, which uses the built-in ScalaCheck function Gen.chooseNum to generate an integer number between 0 and num_zones − 1 for the zone identifier

```
 1  class  Safety_harass_ok
 2    extends  Specification  with ScalaCheck with
 3      DataStreamTLProperty{
 4
 5    def  is  = s2"""Safety ex: ${highDangerNotSafe}"""
 6
 7    // Timeless  incident  generator
 8    def  incidentGen(num_zones : Int,
 9                     min_danger : Double,
10                     max_danger : Double) =
11      for {
12        zone_id  <- Gen.chooseNum[Int](0, num_zones-1)
13        danger   <- Gen.chooseNum[Double](min_danger,
14                                          max_danger)
15      } yield  Incident(zone_id, danger)
16
17    // Testing  function
18    def highDangerNotSafe = {
19      // Generator
20      val  gen = tumblingTimeWindows(Time.hours(1)){
21        WindowGen.always(
22          WindowGen.ofNtoM(15, 50,
23                           incidentGen (10,1.1,10.0)),
24        20)
25      }
26
27      // Property
28      val  property  = Formula.always(
29        Formula.consumeR{
30          case  (input, output) =>
31            output should  foreachElement
32              (_.value._2  != DangerLevel.Safe)
33        }
34      ) during  20
35        groupBy TumblingTimeWindows(Time.hours(1))
36
37      // FlinkCheck  test
38      forAllDataStream[Incident,  (Int,  DangerLevel)]
39      (gen)(harass_max)(property)
40
41    }.set(minTestsOk = 5)
42  }
```

**FIGURE 3. FlinkCheck safety property for HarassMap.**

and a Double between min_danger and max_danger for the danger value—see function incidentGen in line 8 of Fig. 3.

Based on incidentGen, we use FlinkCheck generators to create a stream of timed events organized in 20 tumbling windows of 1 hour, each of them containing between 15 and 50 Incident elements generated by incidentGen. These incidents will have zone identifiers in the range [0–10) and danger values in [1.1–10.0]. The definition of this generator can be found in line 20 of Fig. 3. Note that we store the generator in the variable gen because we will use it later when defining the property to check.

Once we have created the generator, we need to define the property that the stream must satisfy. As the generator creates 20 1-hour-windows of incidents with a danger value greater than 1, the property must express that *always* in 20 windows ($\Box_{20}$) the current window contains only danger levels different from safe ($\lambda^{t}_{(i,o)} not\_safe(o)$, where $not\_safe(o)$ is the formula that checks every element in the current output window $o$ is not Safe). In summary, the $LTL_{ss}$ formula we want to check is $\Box_{20}(\lambda^{t}_{(i,o)} not\_safe(o))$, which is defined in line 28 of Fig. 3. For the operator $\Box_n f$ we use the function Formula.always specifying the number of windows, and complete the property formula with a windowing policy (in this case a tumbling window of 1 hour). The operator $\lambda^{t}_{(i,o)} f$ is provided by several variants of Formula.consume, in this case the variant that accepts an anonymous function that receives an input window and output window for the present instant, and asserts the expected behavior. Note how assertions use the operator should from Specs2 to express in natural language that the output window must satisfy a property. FlinkCheck's foreachElement matcher accepts a Boolean function and uses Flink's operators to efficiently check that all the elements in the specified Flink DataSet satisfy that function (in this case stating that every element in output must have a danger level different from Safe). The elements in the output window are TimedElement objects containing a timestamp and a value, so we first need to extract the value object that will be a (Int, DangerLevel) pair and access the second component with the pattern _.value._2.

Finally, we combine the generator, the test subject, and the $LTL_{ss}$ property to check using the forAllDataStream function described in Sect. III to get a ScalaCheck property that employs a universal quantifier $\forall e \in gen.\ prop(fun(e))$ to relate them: for every element $e$ generated by $gen$, the result $fun(e)$ must satisfy the property $prop$. The property can be found in line 38 of Fig. 3, where gen, harass_max, and property are the generator, function to test, and property previously defined. The result is a ScalaCheck Prop object that can be used with any compatible testing library. For example, we can include this property in a Specs2 specification [24], as shown in Fig. 3. The complete code of this specification can be found in the file Safety_harass_ok at https://git.io/fjiQE. The trait Specification indicates that this is a Specs2 test class—called specification in Specs2 terminology—and the trait ScalaCheck adds compatibility with ScalaCheck. Specs2's is function is the entry point of the specification, and indicates the test functions to run, in this case just highDangerNotSafe—see line 5. Note the Prop object returned by the function highDangerNotSafe in lines 18–41 is modified with minTestsOk = 5 to express that at least 5 generated test cases must satisfy the property formula for the property execution to be considered correct. We can launch this test using Specs2's integration with the build tool *sbt* [26], with the command test:testOnly *Safety_harass_ok*.

The output generated by the execution of the test is shown in Fig. 4. The system also shows additional logging information about the different tested cases and the generated windows of the input and output streams, which we have omitted for conciseness. The summary shows that the property highDangerNotSafe in the specification Safety_harass_ok has passed after successfully checking 5 generated test cases (5 expectations). Any number of required test cases can be specified—see Sect IV-C for

```
sbt: flink −check−examples> test:testOnly
  ∗Safety_harass_ok∗
 ...
WARN DataStreamTLProperty$: Starting execution of
    test case 0
 ...
WARN DataStreamTLProperty$: Completed execution of
    test case 0 with result True
WARN DataStreamTLProperty$: Starting execution of
    test case 1
 ...
WARN DataStreamTLProperty$: Completed execution of
    test case 4 with result True
[info]  Safety_harass_ok + highDangerNotSafe
[info]  Total for specification  Safety_harass_ok
[info]  Finished in 2 minutes 3 seconds, 732 ms
[info]  1 example,5 expectations,0  failure,0  error
[info]  Passed: Total 1, Failed 0, Errors 0, Passed 1
[success]  Total time: 126 s
```

**FIGURE 4.** Execution of the **Safety_harass_ok** property.

```
class  Safety_harass_fail
  extends Specification  with ScalaCheck with
    DataStreamTLProperty{

  def is  = s2"""Failing ex: ${highDangerNotSafe}"""

  def highDangerNotSafe = {

    // Generator of 20 tumbling windows of 1 hour
    // containing between 2 and 5 incidents with
    // values in the range [0.5−10.0]
    val gen2 = tumblingTimeWindows(Time.hours(1)){
      WindowGen.always(
        WindowGen.ofNtoM(2,5,
                         incidentGen (10,0.5,10.0)),
        20)
    }

    // Same property as before
    val property  = always(...)  during  2...

    // FlinckCheck test
    forAllDataStream[Incident, (Int, DangerLevel)]
      (gen2)(harass_max)(property)
  }.set(minTestsOk = 5)
```

**FIGURE 5.** Failing FlinkCheck safety property for HarassMap.

performance results. By default test cases are executed sequentially, but we can optionally set the workers property of the Prop to specify the number of test cases to execute in parallel, in different threads.

The informative messages that FlinkCheck shows when processing tests are useful to detect what has failed. Suppose we want to check the following safety property about harass_max: if the stream of incidents contains only danger values between 0.5 and 10, then no zone can have a Safe danger level. This property must be falsified with the implementation of harass_max presented in Fig. 1, as any maximum danger value between 0.5 and 1 (both included) will generate a Safe danger level. This test is shown in Fig. 5, and the complete code can be found in the file Safety_harass_fail at https://git.io/fjiQg.

```
1  sbt: flink −check−examples> test:testOnly
2    ∗ Safety_harass_fail ∗
3  WARN DataStreamTLProperty$: Starting execution of
4      test case 0
5   ...
6  DEBUG TestCaseContext$ − test case 0: Checking
7      window #0 with timestamp 0
8   ...
9  DEBUG TestCaseContext$ − test case 0: Checking
10     window #1 with timestamp 3600000
11  ...
12 DEBUG TestCaseContext$ − test case 0: Checking
13     window #2 with timestamp 7200000
14  ...
15 DEBUG TestCaseContext$ − test case 0: Checking
16     window #3 with timestamp 10800000
17 DEBUG TestCaseContext$ − test case 0:
18     Time: 10800000 − Input (3 elements)
19 TimedElement(13549799,Incident(5,0.5))
20 TimedElement(14159751,Incident(7,1.522592))
21 TimedElement(13897165,Incident(0,1.0))
22
23 DEBUG TestCaseContext$ − test case 0:
24     Time: 10800000 − Output (3 elements)
25 TimedElement(14399999,(0,Safe))
26 TimedElement(14399999,(5,Safe))
27 TimedElement(14399999,(7,Warning))
28
29 WARN DataStreamTLProperty$: Completed execution of
30     test case 0 with result False
31 [error] x highDangerNotSafe
32 [error]   Falsified after 0 passed tests.
33 [error]   > ARG_0: List(
34     TimedElement(17442,Incident(7,10.0)),
35     TimedElement(2725622,Incident(2,6.859305220235407)),
36     TimedElement(4163859,Incident(0,10.0)), ...,
37     TimedElement(70177063,Incident(9,1.0)))
38 [info]  Safety_harass_fail
39 [info]  Total for specification  Safety_harass_fail
40 [info]  Finished in 8 seconds, 467 ms
41 [info]  1 example, 1 failure, 0 error
42 [error]  Failed: Total 1, Failed 1, Errors 0, Passed 0
43 [error]  Failed tests:
44 [error]   es.ucm.fdi.sscheck.flink.collaborative.
45       Safety_harass_fail
46 [error]  (Test / testOnly) sbt.TestsFailedException:
47     Tests unsuccessful
48 [error]  Total time: 10 s
```

**FIGURE 6.** Execution of the **Safety_harass_fail** property.

The main difference with the class Safety_harass_ok in Fig. 3 is the generator gen2 that creates windows with between 2 and 5 incidents of values in [0.5–10] (we have decreased the number of incidents per window to 2–5 to obtain a more concise and clear output). If we execute this new test as before, FlinkCheck will find with a high probability a stream that falsifies the property, as we can see in Fig. 6. Note that FlinckCheck has generated 3 windows satisfying the property (see lines 6, 9, and 12), but the window #3 starting in line 15 with starting timestamp 10800000 has falsified the property. Concretely, this window contains 3 incidents with their corresponding timestamps:

Incident(5,0.5), Incident(7,1.522592), and Incident (0,1.0) , see lines 19–21. Considering our Flink program harass_max, the zones 0 and 5 would generate a danger level of Safe, as shown in the output of lines 25–27: (0, Safe), (5, Safe), and (7, Warning). At this point FlinkCheck detects the failure of the property and finishes the first test (test 0), showing the complete list of events in the stream (see the lines 33–37, where many elements have been omitted) and also the complete name of the test class that fails (es.ucm.fdi.sscheck.flink.collaborative.Safety_harass_fail in line 44). Note that the test Safety_harass_fail was configured to be tested 5 times (minTestOK = 5) but FlinkCheck has finished the testing process with the first generated stream (test 0) because it already falsified the property.

### B. WRITING A LIVENESS PROPERTY

In this section, we present a more complex liveness property that uses an interesting feature of FlinkCheck: employing a different window size for the generator, the Flink program, and the $LTL_{ss}$ property. This way, programmers and testers can use whichever windowing policies are best suited to their needs. Concretely in this example:

- The generator creates 4 windows of 30 minutes, i.e., 2 hours of incidents.
- The Flink program is the harass_max presented in Sect. II-A, so it processes the incidents each hour.
- The property consumes the original events and those generated by harass_max considering windows of 15 minutes.

Therefore, if at some 15-minute-window we detect an incident with a danger value greater than 8 then that zone must appear as Extreme in the output of the current window or the output of some of the next 3 windows. This situation is graphically shown in Fig. 7, where original events are represented as squares, the events produced by harass_max are represented as circles, and these events are generated at the last instant of the 1-hour-window considered by harass_max. The mentioned liveness property is expressed in $LTL_{ss}$ as $\Box_5(\lambda^t_{(i,o)}\ highest(i) \rightarrow (extreme(o) \vee \Diamond_3\lambda^{t'}_{(i',o')}\ extreme(o')))$. For every evaluation window if it contains zones with a danger value greater than 8—*highest(i)*—then that zone must appear as extreme in the current output—*extreme(o)*—or eventually—$\Diamond_3$—the output $o'$ of some of those future windows must contain that zone as extreme—$extreme(o')$.

Fig. 8 shows the definition of the generator genL (line 1) of 4 windows of 30 minutes containing 4 random Incidents . In this case, we have used the function WindowGen.ofN to generate windows with exactly 4 incidents. We have also modified the invocation to incidentGen to generate incidents with danger values in the complete range [0.0–10.0]. More complex generators, alternating high and low values, could be easily defined using generators like WindowGen.until.

For the liveness property propertyL (line 7 in Fig. 8) we have defined some auxiliary data sets and formulas to clarify and simplify the code. First, we have defined the data set of zone identifiers extremeZones (line 8) to represent those zones with a danger value greater than 8 in the current input window. Similarly, nowExtremeZones (line 15) contains those zones with Extreme levels in the current output window, and the data set futExtremeZones (line 25) contains the Extreme zones in some future output window fut. These data sets are computed using standard Flink transformations: filter (to select those elements with extreme values from a DataSet) and map (to extract the danger value/level from Incident objects or standard tuples). Relying on extremeZones, we define the formula anyExtremeZones (line 12) that checks whether there is any zone with danger value greater than 8 in the current window, i.e., if extremeZones is not empty. Similarly, we create the formula nowExtreme (line 19) that checks whether extremeZones are a subset of nowExtremeZones, i.e., if the zones with a danger greater than 8 are detected as Extreme in the current output window. We also define the formula eventuallyExtreme that checks in the next three 15-minute-windows if extremeZones is a subset of the zones with Extreme level in those future windows (data set futExtremeZones). Note that in this case, we have used the function Formula.eventuallyR with 3 windows to concisely represent the two operators $\Diamond_3\ \lambda^t_{(i',o')}$ of the $LTL_{ss}$ formula. Finally, with these simpler properties, we can define the complete property as Formula.alwaysF containing anyExtremeZones ==> (nowExtreme or eventuallyExtreme). Formula.alwaysF receives an anonymous function from a letter to an $LTL_{ss}$ formula, and as before represents two operators: $\Box_5\lambda^t_{(i,o)}$. The function is configured to consider

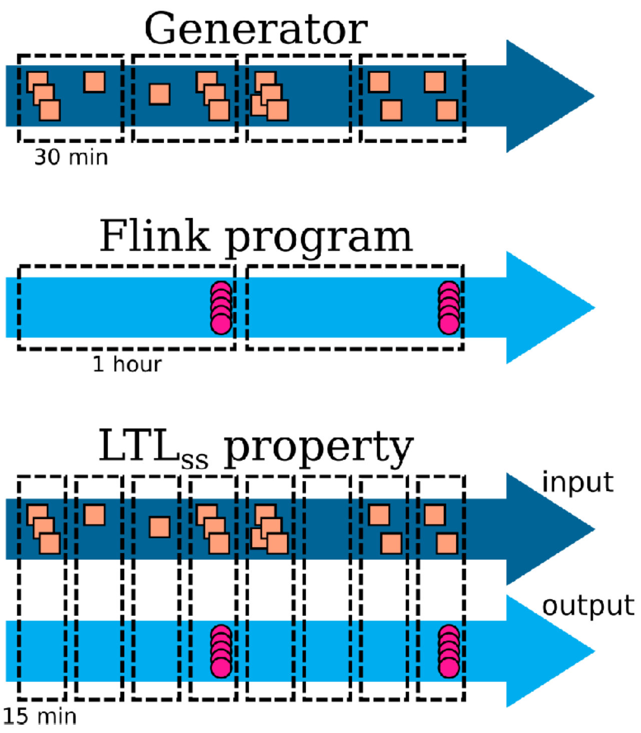


FIGURE 7. Example of different window sizes in the generator (30 minutes), the Flink program (1 hour) and the $LTL_{ss}$ property (15 minutes).

```
val genL = tumblingTimeWindows(Time.minutes(30)){
  WindowGen.always(
    WindowGen.ofN(4, incidentGen(10,0.0,10.0)),
    4)
}

val propertyL = Formula.alwaysF{ case (i, o) =>
  val extremeZones =
    i. filter (_.value.danger > 8)
     .map(_.value.zone_id)

  val anyExtremeZones = Solved{
    extremeZones should beNonEmptyDataSet() }

  val nowExtremeZones =
    o. filter (_.value._2 == Extreme)
     .map(_.value._1)

  val nowExtreme =
    Solved{ extremeZones
      should beSubDataSetOf(nowExtremeZones) }

  val eventuallyExtreme = Formula.eventuallyR {
    case (_,fut) =>
      val futExtremeZones =
        fut. filter (_.value._2 == Extreme)
           .map(_.value._1)
      extremeZones should
        beSubDataSetOf(futExtremeZones)
  } on 3

  anyExtremeZones ==>
    (nowExtreme or eventuallyExtreme)
} during 5
  groupBy TumblingTimeWindows(Time.minutes(15))
```

**FIGURE 8.** Generator and property for the liveness test.

5 windows of 15 minutes using the decorations during and groupBy.

Finally, the liveness FlinkCheck property is defined as in the safety case by using the forAllDataStream function as follows:

```
forAllDataStream[Incident, (Int, DangerLevel)]
  (genL)(harass_max)(propertyL)
```

The complete code can be found in Liveness_harass_ok at https://git.io/fjPCV. As in the previous case, the liveness FlinkCheck property can be included in a Specs2 specification and automate its verification as Scala tests using the standard *sbt* tool—see Liveness_harass_ok for more details.

### C. BENCHMARKS

In order to check the practicability and scalability of our testing framework, we have measured the time required to test the safety and liveness properties presented before using different numbers of windows and events per window—considering the windowing policy of the $LTL_{ss}$ property.

We have used the following methodology for the benchmarks: for each combination of a number of windows and window size, we have launched 100 tests (ScalaCheck's default value) using the local test execution environments included in Flink, that runs the program using the Flink runtime in a single Java Virtual Machine—but using multiple threads and partitioning the data sets and streams across them. We have set workers = 5 to have up to 5 tests running concurrently. The machine used for benchmarks is an 8-core Intel i7 CPU at 3.60GHz with 16GB of memory running Ubuntu 18.04 LTS. Regarding the software, we have used Scala 2.11.8 and Java OpenJDK 64-Bit 1.8.0_222. The results can be found in Tables 3 and 4 for safety and liveness properties, respectively. The main conclusion extracted from these results is that the number of windows has a great impact on the required time, with linear growth. However, increasing the number of events per window (window size) increments very slightly the overall time used for testing the property, although these increments are bigger as the number of windows increases. Fig. 9 shows graphically the growth of testing time w.r.t. the number of windows for the two properties. As the variability caused by the window size is very small w.r.t. the overall time ($< 11\%$), we have used the median time considering all windows sizes (50–350) for plotting the line and represented the minimum and maximum times using error bars. As can be seen, the growth is almost perfectly linear, with best line fits of $\hat{y} = 83.2x + 97.7$ for the safety property and $\hat{y} = 104.8x - 26.9$ for the liveness property. Therefore, increasing by one the number of windows would require approximately 1 minute and 30 seconds of additional time. As a final remark, all the running times are under 50 minutes, which is a reasonable time for an integration test,

**TABLE 3.** Time (seconds) required for executing 100 tests of the safety property (Sect. IV-A) varying the number of windows and the number of incidents per window (window size).

| | | Number of windows | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | 4 | 8 | 12 | 16 | 20 | 24 | 28 |
| Window size | 50 | 424 | 730 | 1051 | 1386 | 1714 | 2006 | 2364 |
| | 100 | 422 | 758 | 1082 | 1390 | 1693 | 2040 | 2382 |
| | 150 | 412 | 752 | 1071 | 1396 | 1747 | 2057 | 2414 |
| | 200 | 427 | 761 | 1097 | 1436 | 1766 | 2099 | 2419 |
| | 250 | 458 | 778 | 1129 | 1478 | 1818 | 2187 | 2540 |
| | 300 | 434 | 777 | 1129 | 1469 | 1826 | 2202 | 2563 |
| | 350 | 430 | 782 | 1163 | 1474 | 1834 | 2220 | 2599 |

**TABLE 4.** Time (seconds) required for executing 100 tests of the liveness property (Sect. IV-B) varying the number of windows and the number of incidents per window (window size).

| | | Number of windows | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | 4 | 8 | 12 | 16 | 20 | 24 | 28 |
| Window size | 50 | 385 | 785 | 1191 | 1615 | 2001 | 2422 | 2845 |
| | 100 | 376 | 806 | 1203 | 1651 | 2060 | 2457 | 2809 |
| | 150 | 370 | 796 | 1221 | 1636 | 2035 | 2413 | 2851 |
| | 200 | 377 | 803 | 1252 | 1677 | 2078 | 2492 | 2894 |
| | 250 | 380 | 803 | 1246 | 1687 | 2040 | 2569 | 2897 |
| | 300 | 387 | 804 | 1250 | 1673 | 2098 | 2545 | 2976 |
| | 350 | 389 | 834 | 1267 | 1671 | 2116 | 2556 | 2946 |

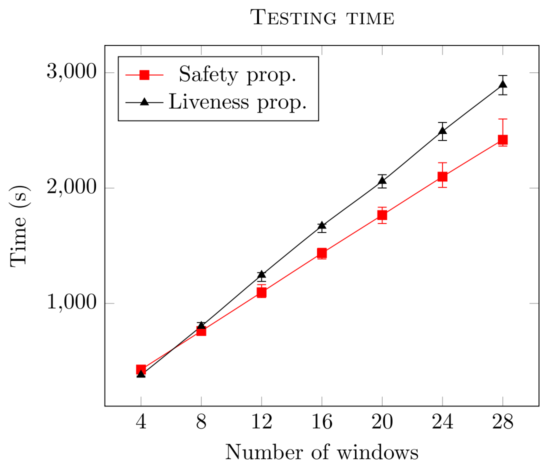


**FIGURE 9. Time required for testing 100 cases of the safety and liveness properties using a different number of windows. Points represent the median time for testing the property with different window sizes, and error bars represent the minimum and maximum time for that number of windows.**

as it is below the default 50 minutes limit for public projects on the popular continuous integration service Travis CI [23].

## V. RELATED WORK

As indicated in the introduction and further developed in [12], the built-in testing features provided by Flink are limited: only unit testing of individual operators, integration testing with a local cluster fake, and some experimental tools for testing checkpointing and state handling [5]. A strong limitation of these testing techniques is that they require the user to manually introduce the input and output streams, hence greatly limiting the number of tested streams and their size. These limitations are found in other stream-processing systems; for example, Apache Spark [27] supports unit testing via the Spark Testing Base library [28]. This library integrates ScalaCheck for Spark but it is only available for Spark core, so it cannot be used with Spark Streaming. Regarding Apache Kafka [29], the documentation [30] indicates that only unit tests are supported, so they are also limited to those inputs that can be created and processed manually by the user.

Beyond the built-in libraries, some other frameworks for particular systems have been proposed. For example, in [31] the authors present a platform for testing automation systems. However, this platform is more focused on testing the underlying network rather than the software properties. In the same line, we find stream generation for multimedia content [32], [33], which deals with the quality of service of the underlying hardware, while the software properties are not checked. In this way, these approaches are complementary to ours and each one will be used in different contexts. Finally, TraceContract [34] 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. Although it can be used for online monitoring of a running system and for evaluating recorded execution traces, it cannot be used to generate test cases and it is not integrated with any standard testing library like Specs2.

To the best of our knowledge, only StreamData [35] for Elixir [36] provides property-based testing for stream-processing systems. StreamData generators return in general a potentially infinite stream that is lazily produced; properties are just Boolean functions that specify, among other values, the maximum size to be extracted from the generator and the number of times the test is executed. This approach is simpler than FlinkCheck because it does not deal with time. On the one hand it might be easier to use because it just works like previous PBT approaches; however, on the other hand, it lacks expressivity for both generators and formulas, hence making it difficult to check complex properties.

It might be argued that FlinkCheck is an evolution of the data-flow approaches for the verification of reactive systems developed in the past decades, exemplified by systems like Lustre [37] and Lutin [38]. In fact, there are some similarities: (a) both process a potentially infinite input stream while generating an output stream and (b) both work with formulas considering both the current state and the previous ones. However, we also find several differences, as explained below.

Lustre is a language for verifying safety properties in reactive systems using random input streams. The random generation provided by FlinkCheck is more refined because it is possible to define particular 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 that FlinkCheck provides, because it supports all Scala features. 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 FlinkCheck. 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"; once we have examined the success of this approach we will try to move into more complex properties. Second, our framework completely separates the input from the output, and hence it is not possible to share information between these streams. Although sharing this information is indeed very important for control systems, we consider that stream processing systems usually deal with external data and hence this relation is not so relevant for the present tool. Finally, note that an advantage of FlinkCheck consists of using the same language for both programming and defining the properties, so an extra specification is not required.

We can also consider runtime monitoring of synchronous systems like Lola [39], 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, FlinkCheck 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.

Regarding previous work, in [7] we presented sscheck, a PBT tool for Spark Streaming [21]. Spark Streaming is based on the notion of Discretized Streams, that are the result of splitting a continuous stream into time-based windows of the same size, called *batches*, which fits easily with $LTL_{ss}$ logic, which uses discrete time. However, we need a bigger effort to adequate the discrete nature of $LTL_{ss}$ to the continuous nature of Flink while preserving its essential features and supporting the richness of its windows, as discussed in Sect III. Currently, we support time-based windows, both tumbling and sliding [1] for test case evaluation, and just tumbling for generators. For this reason and other particular features of Flink, FlinkCheck implementation is completely novel, although we do not need to modify the underlying theory.

Finally, because of the random nature of property-based testing, this kind of tools generate in some cases useless inputs, in the sense that (i) they do not fulfill some semantics restrictions and hence they do not really falsify the given properties or (ii) most of the examples are part of the same ''equivalence class'', while those inputs that are more likely to falsify the property are barely generated. In the first case, there are different techniques [40]–[42] that take into account semantic restrictions to generate better test cases. In the second case, for some programs it is possible to direct the generation towards those values that have a higher probability of falsifying the property [43]. We discuss in the next section how these improvements could be applied to our work.

## VI. CONCLUDING REMARKS AND ONGOING WORK

The main contribution of this paper is FlinkCheck, a property-based testing tool for Apache Flink. We have presented the underlying logic and its instantiation for Apache Flink, the commands implementing this logic, and one example illustrating how to use the tool; more tests based in Apache Flink examples are available in our repository. Finally, some benchmarks analyzing how the tool behaves have been presented. FlinkCheck complements unit testing by providing a framework where key functionality can be tested via properties in bounded temporal logic. This kind of testing can be easily integrated into the quality assurance stage of the software development life cycle, working together with unit and integration testing to provide higher confidence on critical functions. This confidence is essential for modern real-time systems, which take critical decisions that affect human lives.

From the implementation point of view, the current version of FlinkCheck generates the complete input before testing it, so it only supports test cases that fit in the memory of the host that is running the test. Also, FlinkCheck can only run on local mode so some changes would be required for running it on a cluster. As future work, we plan to adapt FlinkCheck to run on a cluster, and also to redesign our generation process so it runs lazily and distributed across the operator subtasks that Flink uses to partition operations, similarly to what Spark Testing Base  [28] does for Spark core. That way we could explore running FlinkCheck on a distributed cluster, generating massive data streams, and use FlinkCheck not only for testing the functional correctness of Flink streaming programs, like we do now, but also for performance testing. For example, FlinkCheck could be used to generate synthetic traffic to stress test a deployment of a Flink program, and experimentally estimate how many hosts are required to handle the expected traffic. These performance tests could also be used to ensure Flink streaming programs are fast enough to handle the expected traffic in production, which is an important concern for stream processing applications [22], and should be checked each time the program is modified. Finally, this kind of performance tests could also be used to perform experimental cost analysis [44].

We can also consider combining our input generator with passive testing frameworks [45]–[47], hence providing traces that would be later analyzed.

Another possible interesting extension would be generating streams by replaying real streams previously stored; in this way we would perform ''delayed'' runtime verification, making sure that our system works as expected for real inputs. This approach can be further improved to provide feedback in real time, which could be complemented with unbounded timeouts for some temporal operators, hence providing a runtime verification tool for potentially infinite streams.

It would be interesting to optimize our random generation procedure, using for example coverage-guided techniques, like the one recently applied in JQF [40], and parametric generators, as described in [41], [42]. Because our test generation procedure takes more time than simpler approaches, improving this stage might be very beneficial for the tool. Coverage-guided techniques (see [48] for details) requires a previous step of code instrumentation for collecting code coverage; then, given an initial set of inputs, they are randomly mutated [49]; those mutants that increase the coverage are saved for later mutation. Code coverage can be improved by checking the validity of the arguments [41], [42], hence biasing the generation towards semantically valid arguments. In our case, it would be interesting to use coverages designed for functional programs, like the proposed in [50], while validity fuzzing can take advantage of Scala preconditions (assert, assume, require, and ensuring). Regarding parametric generators, they are a modification of standard random generators that are turned deterministic by fixing an underlying stream of bits. Mutations in this stream lead to structural mutations of semantically valid inputs, obtaining in this way a better coverage. In our case, we would need to transform ScalaCheck generators, which are the underlying generators of FlinkCheck, in order to obtain this kind of deterministic generators.

Another interesting optimization technique is targeted property-based testing [43], which uses utility functions to search for those values that are most likely to falsify the property. The weak point of this technique is that it is only applicable when such utility functions are available. In order to implement this mechanism in FlinkCheck we need to study first the kind of programs that would benefit from it. Once new examples and real systems are tested with FlinkCheck we will decide the most appropriate direction to optimize generators.

Currently, FlinkCheck does not support counterexample minimization via shrinking [9], which is a typical feature in PBT, but we consider adding this feature in the future, especially after the promising performance results, that suggest the minimization process would not be too slow to use it in practice. So far FlinkCheck can only be used for testing stream transformations, but we also contemplate adding support for testing checkpointing and state handling in Flink. We could also support other windowing policies for generators and property formulas, like session-based windows. Finally, our property formulas support the consume $LTL_{ss}$ operation, but FlinkCheck does not include a generator combinator for consume; implementing a consume generator could be interesting to generate streams where letters can depend on the value of some previous letters, in the style of Lutin [38].

## REFERENCES

[1] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, “Apache flink: Stream and batch processing in a single engine,” *IEEE Data Eng. Bull.*, vol. 38, no. 4, pp. 28–38, 2015.

[2] Amazon. *Amazon Kinesis Data Analytics*. Accessed: Sep. 16, 2019. [Online]. Available: https://aws.amazon.com/kinesis/data-analytics/

[3] Alibaba. *Realtime Compute*. Accessed: Sep. 16, 2019. [Online]. Available: https://www.alibabacloud.com/products/realtime-compute

[4] Huawei. *Huawei Cloud Stream Service*. Accessed: Sep. 16, 2019. [Online]. Available: https://www.huaweicloud.com/en-us/product/cs.html

[5] Apache. *Flink Testing Documentation*. Accessed: Mar. 27, 2019. [Online]. Available: https://ci.apache.org/projects/flink/flink-docs-stable/dev/stream/testing.html

[6] G. Meszaros, *xUnit Test Patterns: Refactoring Test Code*. London, U.K.: Pearson, 2007.

[7] A. Riesco and J. Rodríguez-Hortalá, “Property-based testing for Spark Streaming,” *Theory Pract. Logic Program.*, vol. 19, no. 4, pp. 574–602, 2019.

[8] S. Shamshiri, J. M. Rojas, G. Fraser, and P. McMinn, “Random or genetic algorithm search for object-oriented test suite generation?” in *Proc. Genetic Evol. Comput. Conf.*, 2015, pp. 1367–1374.

[9] K. Claessen and J. Hughes, “Quickcheck: A lightweight tool for random testing of Haskell programs,” in *Proc. 5th ACM SIGPLAN Int. Conf. Funct. Program. (ICFP)*, New York, NY, USA, 2000, pp. 268–279 [Online]. Available: http://doi.acm.org/10.1145/351240.351266

[10] T. Akidau, S. Chernyak, and R. Lax, *Streaming Systems: The What, Where, When, and How of Large-Scale Data Processing*. Newton, MA, USA: O’Reilly Media, 2018.

[11] T. White, *Hadoop: The Definitive Guide*. Newton, MA, USA: O’Reilly Media, 2012.

[12] F. Hueske and V. Kalavri, *Stream Processing with Apache Flink: Fundamentals, Implementation, and Operation of Streaming Applications*. Newton, MA, USA: O’Reilly Media, 2019. [Online]. Available: https://books.google.com/books?id=64GHAQAACAAJ

[13] R. Nilsson, *ScalaCheck: The Definitive Guide*. Walnut Creek, CA, USA: Artima, 2014. [Online]. Available: https://books.google.es/books?id=AOvcoQEACAAJ

[14] M. Odersky, P. Altherr, V. Cremet, I. Dragos, G. Dubochet, B. Emir, S. McDirmid, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, L. Spoon, and M. Zenger, “An overview of the scala programming language,” EPFL, Lausanne, Switzerland, Tech. Rep. LAMP-REPORT-2006-001, 2004.

[15] R. Chiao, E. Ghozlan, A. Fahmy, and S. Gad. *HarassMap | Stop Sexual Harassment*. [Online]. Available: https://harassmap.org

[16] S. Yuce, N. Agarwal, R. Wigand, and R. Robinson, “Cooperative networks in interor-ganizational settings: Analyzing cyber-collective action,” in *Proc. Workshop Multiagent Interact. Syst Netw. (MAIN)*, 2013, pp. 1–9.

[17] T. Arts, J. Hughes, J. Johansson, and U. Wiger, “Testing telecoms software with quviq quickcheck,” in *Proc. ACM SIGPLAN Workshop Erlang*, M. Feeley and P. W. Trinder, Eds. 2006, pp. 2–10. doi: 10.1145/1159789.1159792.

[18] M. Papadakis and K. Sagonas, “A PropEr integration of types and function specifications with property-based testing,” in *Proc. ACM SIGPLAN Erlang Workshop*. New York, NY: ACM Press, Sep. 2011, pp. 39–50.

[19] C. Amaral, M. Florido, and V. S. Costa, “PrologCheck—Property-based testing in prolog,” in *Functional and Logic Programming*, M. Codish and E. Sumii, Eds. Cham, Switzerland: Springer, 2014, pp. 1–17.

[20] P. Blackburn, J. van Benthem, and F. Wolter, Eds., *Handbook of Modal Logic*. Amsterdam, The Netherlands: Elsevier, 2006.

[21] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, “Discretized streams: Fault-tolerant streaming computation at scale,” in *Proc. 24th ACM Symp. Oper. Syst. Princ.*, 2013, pp. 423–438.

[22] A. Bifet, G. Holmes, R. Kirkby, and B. Pfahringer, “MOA: Massive online analysis,” *J. Mach. Learn. Res.*, vol. 11, pp. 1601–1604, May 2010.

[23] Travis CI Team. (2019). *Travis CI Documentation: Customizing the Build, Build Timeouts*. [Online]. Available: https://docs.travis-ci.com/user/customizing-the-build/#build-timeouts

[24] E. Torreborre. *Specs$^2$: Software Specifications for Scala*. [Online]. Available: http://etorreborre.github.io/specs2/

[25] Ververica. (2019). *Apache Flink Training*. [Online]. Available: https://training.ververica.com

[26] *SBT—The Interactive Build Tool*. Accessed: Sep. 16, 2019. [Online]. Available: https://www.scala-sbt.org/

[27] Apache Spark Team. (2016). *Spark Programming Guide*. [Online]. Available: https://spark.apache.org/docs/latest/programming-guide.html

[28] H. Karau. (2015). *Spark-Testing-Base*. [Online]. Available: http://spark-packages.org/package/holdenk/spark-testing-base

[29] N. Narkhede, G. Shapira, and T. Palino, *Kafka—The Definitive Guide*. Newton, MA, USA: O’Reilly Media, 2017.

[30] *Testing Kafka Streams*. Accessed: Sep. 16, 2019. [Online]. Available: https://kafka.apache.org/20/documentation/streams/developer-guide/testing.html

[31] R. Eidenbenz, A. Moga, T. Sivanthi, and C. Franke, “MARS: A flexible real-time streaming platform for testing automation systems,” in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, D. Atienza and G. D. Natale, Eds., Mar. 2017, pp. 518–523.

[32] J. Kim, J. Yang, and D. Kwon, “Test stream generation for digital cable UHD broadcasting standard,” in *Proc. IEEE 6th Int. Conf. Consum. Electron. Berlin (ICCE-Berlin)*, Sep. 2016, pp. 218–221.

[33] Z. Akhtar and T. H. Falk, “Audio-visual multimedia quality assessment: A comprehensive survey,” *IEEE Access*, vol. 5, pp. 21090–21117, 2017. doi: 10.1109/ACCESS.2017.2750918.

[34] H. Barringer and K. Havelund, “TraceContract: A scala DSL for trace analysis,” in *Proc. 17th Int. Symp. Formal Methods (FM)*, in Lecture Notes in Computer Science, vol. 6664, M. J. Butler and W. Schulte, Eds. Berlin, Germany: Springer, 2011, pp. 57–72.

[35] *StreamData: Property-Based Testing and Data Generation for Elixir*. Accessed: Sep. 16, 2019. [Online]. Available: https://github.com/whatyouhide/stream_data

[36] S. Juric, *Elixir in Action*. Shelter Island, NY, USA: Manning, 2019.

[37] N. Halbwachs, *Synchronous Programming of Reactive Systems* (The Springer International Series in Engineering and Computer Science), no. 215. Norwell, MA, USA: Kluwer, 1992.

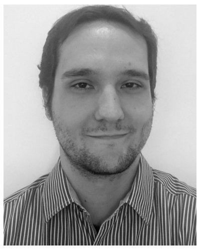
[38] P. Raymond, Y. Roux, and E. Jahier, “Lutin: A language for specifying and executing reactive scenarios,” *EURASIP J. Embedded Syst.*, vol. 2008, 2008, Art. no. 753821. doi: 10.1155/2008/753821.

[39] B. D’Angelo, S. Sankaranarayanan, C. Sánchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra, and Z. Manna, “LOLA: Runtime monitoring of synchronous systems,” in *Proc. 12th Int. Symp. Temporal Represention Reasoning (TIME)*, Jun. 2005, pp. 166–174.

[40] R. Padhye, C. Lemieux, and K. Sen, "JQF: Coverage-guided property-based testing in Java," in *Proc. 28th ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, Jul. 2019, pp. 398–401.
[41] R. Padhye, C. Lemieux, K. Sen, M. Papadakis, and Y. Le Traon, "Validity fuzzing and parametric generators for effective random testing," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng., Companion (ICSE-Companion)*, J. M. Atlee, T. Bultan, and J. Whittle, Eds., May 2019, pp. 266–267.
[42] R. Padhye, C. Lemieux, K. Sen, M. Papadakis, and Y. Le Traon, "Semantic fuzzing with zest," in *Proc. 28th ACM SIGSOFT Int. Symp. Softw. Test. Anal.*, Jul. 2019, pp. 329–340.
[43] A. Löscher and K. Sagonas, "Targeted property-based testing," in *Proc. 26th ACM SIGSOFT Int. Symp. Softw. Test. Anal. (ISSTA)*, New York, NY, USA, Jul. 2017, pp. 46–56.
[44] C. C. McGeoch, *A Guide to Experimental Algorithmics*. Cambridge, U.K.: Cambridge Univ. Press, 2012.
[45] M. G. Merayo, R. M. Hierons, and M. Núñez, "A tool supported methodology to passively test asynchronous systems with multiple users," *Inf. Softw. Technol.*, vol. 104, pp. 162–178, Dec. 2018. doi: 10.1016/j.infsof.2018.07.013.
[46] R. M. Hierons, M. G. Merayo, and M. Núñez, "An extended framework for passive asynchronous testing," *J. Log. Algebr. Methods Program.*, vol. 86, no. 1, pp. 408–424, Jul. 2017. doi: 10.1016/j.jlamp.2016.02.004.
[47] A. R. Cavalli, C. Gervy, and S. Prokopenko, "New approaches for passive testing using an extended finite state machine specification," *Inf. Softw. Technol.*, vol. 45, no. 12, pp. 837–852, Sep. 2003. doi: 10.1016/S0950-5849(03)00063-6.
[48] V. J. M. Manes, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," 2018, *arXiv:1812.00140*. [Online]. Available: https://arxiv.org/abs/1812.00140
[49] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Trans. Softw. Eng.*, vol. 37, no. 5, pp. 649–678, Sep./Oct. 2011. doi: 10.1109/TSE.2010.62.
[50] S. Fischer and H. Kuchen, "Systematic generation of glass-box test cases for functional logic programs," in *Proc. 9th ACM SIGPLAN Int. Conf. Princ. Pract. Declarative Program.*, Jul. 2007, pp. 63–74.

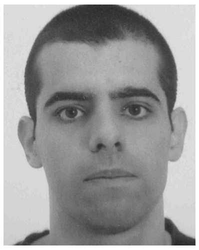
**CRISTINA VALENTINA ESPINOSA** was born in Madrid, Spain, in 1993. She received the B.S. and M.S. degrees in computer science from the Universidad Complutense de Madrid, Spain, in 2015 and 2019, respectively.

Her main research interests include big data and artificial intelligence. Her goals aim to make use of these fields to create predictive models that make people's lives easier.



**ENRIQUE MARTIN-MARTIN** was born in Madrid, Spain, in 1984. He received the B.S., M.S., and Ph.D. degrees in computer science from the Universidad Complutense de Madrid, Spain, in 2007, 2009, and 2012, respectively.

He was an Assistant Professor with the Department of Software Systems and Computing, Universidad Complutense de Madrid, from 2007 to 2012, and an Interim Professor, from 2013 to 2016, where he has been a Ph.D. Assistant Professor, since 2016. He has authored more than 25 articles presented in international conferences and journals. He teaches postgraduate courses on big data, machine learning, and information retrieval. He has coauthored two text books about big data and data analysis. His research interests include static analysis of concurrent and multiparadigm programs, as well as declarative debugging and big data analysis.



**ADRIÁN RIESCO** was born in Madrid, Spain, in 1982. He received the B.S., M.S., and Ph.D. degrees in computer science from the Universidad Complutense de Madrid, Spain, in 2005, 2007, and 2011, respectively.

He was an Interim Professor, from 2011 to 2015, and a Ph.D. Assistant Professor, from 2015 to 2018, with the Department of Software Systems and Computation, Universidad Complutense de Madrid, where he has been an Associate Professor, since 2018. He has authored more than 50 articles. His research interests include formal methods, rewriting logic, declarative debugging, testing, and big data analysis.

Dr. Riesco was a recipient of the 2005 Degree Award in computer science of the UCM. He has collaborated as a Program Committee Member and a Reviewer for many international conferences and journals.

**JUAN RODRÍGUEZ-HORTALÁ** was born in Madrid, Spain, in 1981. He received the B.S., M.S., and Ph.D. degrees in computer science from the Universidad Complutense de Madrid, Spain, in 2005, 2007, and 2010, respectively.

He was a Teaching Assistant, from 2008 to 2011, and a Ph.D. Assistant Professor, from 2011 to 2012, with the Department of Software Systems and Computation, Universidad Complutense de Madrid, where he has been a Software Developer, since 2012, and a Ph.D. Assistant Professor with the Department of Software Engineering and Artificial Intelligence, since 2019. He has authored more than 25 articles. His research interests include programming language semantics, logic, type systems, declarative programming, and testing.