StreamGen: Model-Driven Development of Distributed Streaming Applications

MICHELE GUERRIERO, Politecnico di Milano - DEIB, Italy
DAMIAN ANDREW TAMBURRI, Technical University of Eindhoven - JADS, The Netherlands
ELISABETTA DI NITTO, Politecnico di Milano - DEIB, Italy

Distributed streaming applications, i.e., applications that process massive streams of data in a distributed fashion, are becoming increasingly popular to tame the velocity and the volume of *Big Data*. Nevertheless, the widespread adoption of data-intensive processing is still limited by the non-trivial design paradigms involved, which deal with the unboundedness and volume of involved data streams, and by the many distributed streaming platforms, each with its own characteristics and APIs. In this paper, we present `StreamGen`, a Model-Driven Engineering tool to simplify the design of such streaming applications and automatically generate the corresponding code. `StreamGen` is able to automatically generate fully working and processing-ready code for different target platforms (e.g., Apache Spark, Apache Flink). Evaluation shows that: (i) `StreamGen` is general enough to model and generate the code, offering comparable performance against a preexisting similar and well-known application; (ii) the tool is fully-compliant with streaming concepts defined as part of the Google Dataflow Model; (iii) users with little computer science background and limited experience with big data have been able to work with `StreamGen` and create/refactor an application in a matter of minutes.

CCS Concepts: • **Software and its engineering** → **Application specific development environments**; *Software design engineering*; *Rapid application development*; *Software prototyping*.

Additional Key Words and Phrases: Model-Driven Engineering, Streaming Applications, Big Data Architectures

ACM Reference Format:
Michele Guerriero, Damian Andrew Tamburri, and Elisabetta Di Nitto. 2018. StreamGen: Model-Driven Development of Distributed Streaming Applications. 1, 1 (June 2018), 31 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Over the last years we have witnessed a lot of interest in *data-intensive technologies and applications* to exploit Big Data and extract its hidden business intelligence. The complexity of extracting such intelligence is typically associated with the so called *3 Vs*, i.e., the volume, velocity, and variety of the data to be analysed. Specific Big Data scenarios can own one or more of these characteristics. In particular, for those scenarios in which volume and

Authors' addresses: Michele Guerriero, Politecnico di Milano - DEIB, Via Golgi 42, Milano, Italy; Damian Andrew Tamburri, Technical University of Eindhoven - JADS, Sint Janssingel 92 5211 DA, s'Hertogenbosch, The Netherlands; Elisabetta Di Nitto, Politecnico di Milano - DEIB, Via Golgi 42, Milano, Italy.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
XXXX-XXXX/2018/6-ART $15.00
https://doi.org/10.1145/1122445.1122456

velocity are primary concerns, distributed streaming platforms come into play. These allow designing *streaming applications*, i.e., applications that process potentially infinite streams of data, continuously and rapidly updating useful outputs (statistics, predictions etc.). Such applications are then reliably executed in a distributed fashion on large clusters of computational nodes, in order to deal with the huge volume of incoming data. Streaming applications are gaining momentum due to their high business potential [1].

Unfortunately, according to research done at Capgemini [9] and to Gartner Inc [2], by 2016, only 14% of organizations have achieved full-scale production for their applications exploiting Big Data. There are several reasons behind this problem. Among the others, 1) the non-trivial, dataflow-based design paradigms that need to be employed for developing distributed streaming applications [1, 2] and 2) the technology heterogeneity due to the many existing distributed streaming platforms (e.g., Apache Storm, Apache Spark Streaming and Apache Flink), each coming with its own characteristics and APIs [19]. For example, Flink offers a streaming-first engine in which the incoming data is processed as soon as it arrives, which is the optimal choice when computation latency is a primary performance concern. Spark, instead, generally provides better throughput at the expense of latency as it relies on a micro-batch engine, i.e., a stream of data is discretized by grouping incoming data items into batches before being processed. Moreover, Spark allows different type of workloads (batch, streaming, interactive) to coexist within the same application, something which is not possible in Flink. For example, Apache Kafka and Apache Storm are built around other design principles and goals. Such heterogeneity currently imposes severe difficulties throughout the application lifecycle (e.g., time-consuming trade-off analysis, re-engineering of the application in case of migration to different platform versions).

Researchers and practitioners are approaching the problem from different angles, ranging from studying and defining the semantics of stream processing engines, to defining conceptual and programming models which capture the main abstractions behind distributed streaming applications [1, 2]. Most notably, the Google Dataflow Model provides the semantic foundations for streaming applications and acts as a reference model for the capabilities of modern stream processors. Indeed, the many available engines can be benchmarked against the Google Dataflow Model [3] and Apache Beam, which offers an implementation of the model, allows to program streaming applications which can then be executed on multiple platforms (e.g., Spark, Flink, etc.), as we will discuss more in detail in Section 8. Although the Google Dataflow Model and its implementation Apache Beam represent important steps towards defining a common semantics for modern streaming applications and dealing with the technology heterogeneity, it is still not easy to speedily prototype streaming applications on multiple platforms using Beam because it still requires non-trivial programming expertise.

Our research assumption is that Model-Driven Engineering (MDE) has some potential in addressing the issues we have discussed so far about designing and developing modern streaming applications. In fact, among the promises of MDE are: 1) simplifying the understanding of a system by means of abstract and graphical modeling, 2) dealing with the technology heterogeneity and 3) fostering productivity and fast prototyping [7, 16]. Moreover, we note that a streaming application can be easily represented as dataflow graph, which lends itself to be modeled using graphical approaches, with advantages in terms of visualizing, understanding and reasoning about the application.

[1] https://www.forbes.com/sites/forbestechcouncil/2018/05/11/the-competitive-advantage-of-streaming-analytics/#50dc0b2a5957

[2] https://www.gartner.com/newsroom/id/3466117

[3] https://beam.apache.org/documentation/runners/capability-matrix/

In this paper we present the design and development of `StreamGen` [4], a model-driven tool that relies on the conceptual similarities between the various streaming platforms to simplify and speed-up the design, development, and operation of distributed streaming applications, while overcoming the problem of begin locked-in with specific execution engines. `StreamGen` offers two main advances:

- A novel Domain-Specific Modeling Language (DSML) in the form of a UML profile [18, 28] that we called *StreamUML*, which allows designers to model a streaming application as a dataflow graph using standard UML modeling and in particular Composite Structure Diagrams. StreamUML provides the main abstractions required to model a distributed streaming application thus disengaging developers from knowing details of the specific target platforms, while still letting platform-specific customizations to be configured.
- A code generation module, called *StreamCGM*, which transforms profiled UML models into the application code for the specific target platform of choice. Currently StreamCGM includes 2 code generators for two of the most popular open source distributed streaming platforms, namely Apache Flink and Apache Spark. Nevertheless, it is designed in a modular fashion so to be extensible.

`StreamGen` has been previously presented in a two-pages tool demo paper published in the proceedings of the 10th International Workshop on Modelling in Software Engineering [14]. While the demo paper was focusing solely on demonstrating the tool in action, in this paper we fully elaborate on its design, implementation and research contributions.

The rest of the paper is organized as follows. Section 2 provides some background information on the structure of modern streaming applications. Section 3 presents an overview of the `StreamGen` approach; Section 4 presents StreamUML and shows how it can be used with a running example; Section 5 discusses the implementation of StreamCGM; Section 6 presents the evaluation we have carried out for `StreamGen`; Section 7 discusses the threat to validity; Section 8 presents related work; finally Section 9 concludes the paper and outlines the future work.

2 BACKGROUND

2.1 Context and Main Notions

In this Section, we provide the necessary background to understand how modern distributed streaming applications are developed and executed.

In terms of application development, many data-intensive *frameworks* for developing distributed streaming application have evolved a lot over the last decade. One of the first and most popular ones is Apache Storm[5], known for its reliable and distributed stream processing capabilities, which, however, provided relatively limited development capabilities. For example, there was no support for *windowing semantics*—that is, the operational semantics which allow data bursts to be processed in windows [15] without loosing data partitions or timeliness properties—that needed to be explicitly hardcoded by application developers. Moreover, the level of abstraction was very low, as applications were expressed in terms of their *topology* of *data operators*, which basically corresponded to the way the application was actually executed, with no intermediate abstractions between development and operations in support of developers' coding quantities and quality.

[4] https://github.com/MicheleGuerriero/streamgen
[5] https://storm.apache.org/

From that time, more advanced frameworks have been developed, offering more streaming features out-of-the-box (e.g., windowing) along with higher-level abstractions aimed at simplifying application development. In particular, the majority of modern frameworks for developing distributed streaming application offer *functional programming* models, which greatly simplify the development process (see Section 4.1). Examples of such frameworks are Apache Flink[6], Apache Spark[7], and Apache Kafka[8].

Ultimately, the Google Dataflow Model [2] represents an attempt to unify the various, similar abstractions provided by the many available frameworks into a comprehensive model and corresponding APIs (i.e., Apache Beam[9]).

2.2 Scope and Modelling Semantics Demarcations

In terms of scope of our research solution in terms of deployment and operation of distributed streaming applications, although different engines are based on different paradigms, such as massage passing or micro-batches, the basic idea is to have applications translated into topologies, or Direct Acyclic Graphs (DAGs), in which operators are individually deployable components, in order to enable fault-tolerance mechanisms. Additionally, operators are typically parallelized, meaning that there are multiple running instances of an operator, each of which can be seen as a separate process, which can be potentially running on its own environment.

It is important to highlight how, according to these deployment strategies, a single streaming application is actually an ensemble of independent and individually deployable components, although this might be not immediately evident from the application design. The distributed streaming framework is responsible to make decisions on how to distribute the parallel calculus, with specific indications under which the input shall be subdivided, whereas the scheduler will then proceed to schedule jobs and ensure their execution.

The focus and scope of StreamGen—and similar technologies—shall be to support the roles and responsibility of defining the Dataflow to be enacted in the target streaming framework. Such support shall concentrate on rapid prototyping, i.e., defining the aforementioned dataflow without delving in the operational semantics and/or configuration details for such technologies.

3 STREAMGEN OVERVIEW

`StreamGen` focuses on the following two requirements:

- *To provide a platform-independent and graphical modeling language for streaming applications.* This language shall capture the abstractions that are relevant in the streaming context and shall assist the development process by enabling appropriate consistency checking of application designs. Moreover, it shall provide a graphical modeling notation which is easy to learn and use.
- *To enable the fast prototyping of distributed streaming applications over different platforms.* More specifically, `StreamGen` shall provide development teams with means to shorten the time needed for prototyping and experimenting with a streaming application by automating its code generation for different target platforms.

[6]https://flink.apache.org/
[7]https://spark.apache.org/
[8]https://kafka.apache.org/
[9]https://beam.apache.org/

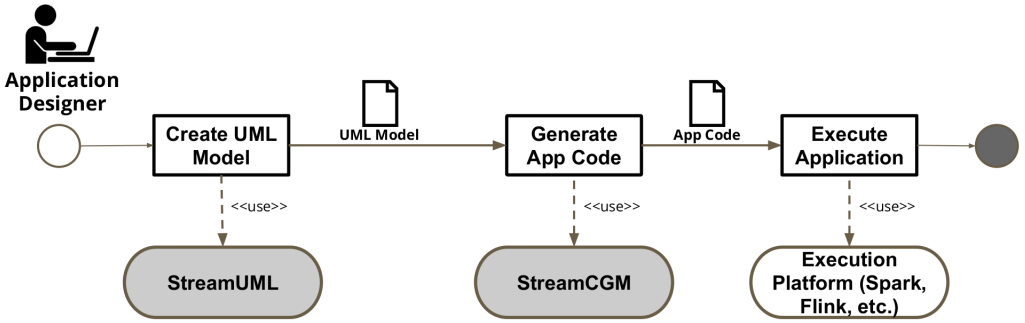


Fig. 1. Overview of the StreamGen usage workflow.

Figure 1 provides an overview of the `StreamGen` workflow from the perspective of an application designer. The white and grey circles represent the beginning and the end of the workflow respectively. Square boxes represent activities with solid arrows connecting activities in order to build up the workflow. Document symbols placed on solid arrows represent artifacts that are produced at the end of a given activity and passed as input to the next one. Dashed arrows connect activities to the tools used by the application designer in order to carry out them. Tools are represented with rounded boxes (those that are original contributions of this research are marked in grey).

The designer exploits UML modeling and in particular Composite Structure Diagrams enriched with the StreamUML profile (implemented in Eclipse Papyrus), to create a model of her streaming application in a platform-independent way, both in terms of its dataflow graph of computation as well as more advanced aspects, such as its parallelization and time semantics. During this modeling activity, the designer is supported by the validation mechanisms defined in StreamUML via OCL constraints. Then, the designer enacts the code generation using StreamCGM (implemented in Eclipse Acceleo), which takes as input the created UML model and produces the application code. Finally, the designer or any other user of the streaming application can execute the generated code, using the target execution platform of choice (e.g., Spark or Flink).

In the following sections, we are going to detail the various components of `StreamGen` and discussing the rationale of the design choices we made. To make the presentation easy to be followed, we will use the *StreamingWordCount* application as a running example. This is essentially the "Hello World" in the context of streaming applications and exploits some of the most commonly used streaming features. In particular, the application continuously receives pieces of text via a socket connection, observes them for a time interval of 3 minutes and computes, through parallel processing, the number of occurrences of each distinct word seen during the last time interval. Such results are eventually stored into a local file.

4 MODELING LANGUAGE

We defined StreamUML following the state of the art in UML profile construction [18, 28]. We first defined the domain model for streaming applications relying on our experience with several streaming platforms (Section 4.1), and then, based on this, we designed the StreamUML profile. Section 4.2 provides an overview of this profile, Section 4.3 presents the details of its implementation in UML.

4.1 A Domain Model for Streaming Applications

Our domain model stems from our own experience prototyping applications with several, widely-known distributed streaming platforms, namely, Apache Storm, Apache Spark, Apache Flink, Apache Beam, and Apache Kafka. Additionally, it relies on the experience carried out by DICE project team we were part of, which has devised domain models for data-intensive applications (which include streaming ones) developed using various platforms, among which, Apache Storm and Apache Spark [21, 22]. The joint experience in modeling and prototyping these kinds of applications for different available platforms drove us towards the definition of a common domain model presented hereby. In particular, we noticed that all the distributed streaming platforms we have experimented with offer a common set of features, although through (sometimes very) different APIs. For example, let us consider the snippets of code reported in listing 1, 2 and 3, which are extracted from the implementations of the StreamingWordCount application for the Apache Beam, Spark and Flink platforms, respectively. It is easy to note that, although with different APIs, the three code fragments share a common set of aspects. They all begin by specifying a text file to be read; the output of this first operation is then streamed through a sequence of operations; in all the cases there is a notion of observation window and of key-values pairs stream.

```
1  PCollection<KV<String, Integer>> wordCounts =
2  pipeline
3    .apply(TextIO.read().from(options.getInputFile()))
4    .apply(Window.into(FixedWindows.of(Duration.standardMinutes(2))))
5    .apply(new WordCount.CountWords());
```

Listing 1. Core sequence of operations for the StreamingWordCount application – Apache Beam

```
1  JavaPairDStream<String, Integer> counts =
2    sc.textFile("...")
3      .flatMap(s -> Arrays.asList(s.split(" ")).iterator())
4      .mapToPair(word -> new Tuple2<>(word, 1))
5      .reduceByKeyAndWindow((a, b) -> a + b, Durations.minutes(2));
```

Listing 2. Core sequence of operations for the StreamingWordCount application – Apache Spark

```
1  DataStream<Tuple2<String, Integer>> counts =
2    env.readTextFile("...")
3    .flatMap(new LineSplitter())
4        .keyBy(0)
5        .timeWindow(Time.minutes(2))
6        .apply(new WordCounter());
```

Listing 3. Core sequence of operations for the StreamingWordCount application – Apache Flink

In general, a streaming application can be conceived as a dataflow graph in which edges represent data streams and nodes represent *operators* which produce, transform and store data streams. Each data stream is essentially an infinite set of *tuples*. A tuple of a data stream is typically structured into a finite set of fields defining the *schema* of the data stream. Given that the operators are independent from each other, they can execute concurrently (potentially on different machines) as soon as their input streams are available, thus enabling *task parallelism*; moreover, *data parallelism* can also be exploited by means of data partitioning, i.e., splitting a given data stream into a number of partitions, each of which is then assigned to a given replica of an operator in order to be processed. To deal with the infiniteness of data streams, usually, a notion of *windowing* is provided to chop up a data stream into finite, sequentially-processed pieces (typically, but not always, along temporal boundaries).

Based on this understanding, we defined our domain model whose core part is shown in Figure 2. The main concepts are the following:
DataStream: data streams are first-class citizens in streaming applications. They connect the various operators of an application, thus defining the corresponding dataflow graph. Each data stream conveys tuples that conform to a specific and fixed schema. This is defined by the *conveyedDatatype* attribute of DataStream.
StreamingOperator: the nodes of the dataflow graph representing a streaming application are operators producing, transforming or storing data streams. An operator can be parallelized, i.e., can be created in multiple instances, by setting the value of the *parallelism* property.
DataSource: it is a specific type of operator which feeds the application with an input data stream. Data sources are the only nodes in the dataflow graph not having input streams, since they are, from a topological point of view, the beginning of a graph. Specific types of data sources, which extend the abstract definition, are then used to model the real world data sources from where the data are actually fetched/received (e.g., sensors, web APIs, etc.).
Transformer: it is a specific type of operator responsible for transforming some input streams to some output streams. This very generic definition imposes no limitation on the actual transformer logic (which can be any) nor on characteristics of the input and output data streams. Specific types of transformers, which extend the abstract definition, have their own semantics and requirements. For example, a given transformer may require a single input stream and perform the cumulative sum over the values of a field.
DataSink: it is a particular type of operator responsible for emitting or storing a given data stream to some external system. Data sinks are the only nodes in the dataflow graph not having output streams, since they are, from a topological point of view, the end of a graph. Specific types of data sinks, which extend the abstract definition, are then used to model various external systems (e.g., databases, dashboards, etc.).
PartitionedStream: as previously mentioned, one of the main feature of distributed streaming applications is that their processing can be parallelized by *partitioning* a stream so that it can be processed in parallel by multiple instances of an operator. The way a stream is partitioned depends on the application logic. For instance it can be partitioned randomly over a given number of partitions (i.e., tuples are assigned to the available partitions according to a random distribution) or based on a key attribute (i.e., tuples are grouped together forming partitions according to the specified key). Other partitioning strategies rely on specific scheduling algorithms (e.g., Round-robin) in order to dispatch tuples to different partitions.
WindowedStream: as previously mentioned, in order to deal with the unboundedness of data streams, these can be *windowed* by means of specific windowing policies. It is worth to note that windowing a stream differs from partitioning it. In fact, while partitioning deals with defining multiple sub-streams which can then be processed in parallel, windowing relates to the way a *single* stream (or a stream partition) is processed, i.e., by splitting it into chunks (or windows), which are processed sequentially.
WindowingPolicy: a data stream can be windowed according to different windowing policies. For instance, in a *CountWindow* the *size* is based on the tuples count (i.e., every window contains a fixed number of tuples), while in a *TimeWindow* it is based on temporal boundaries (i.e., every window contains all the tuples arrived within a time interval of fixed duration). Moreover, subsequent windows can overlap, in which case the windowing policy is said to be *sliding* (both Count and Time Windows have the *slideSize* attribute that defines the size of the slide). Other types of windows can be defined based on some characteristic of the data

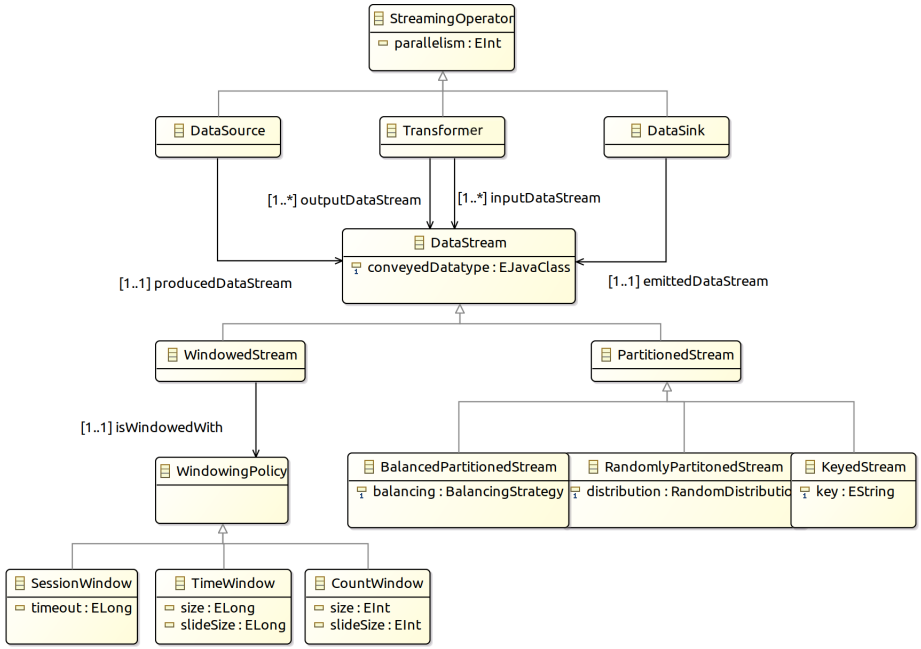


Fig. 2. Domain model for streaming applications.

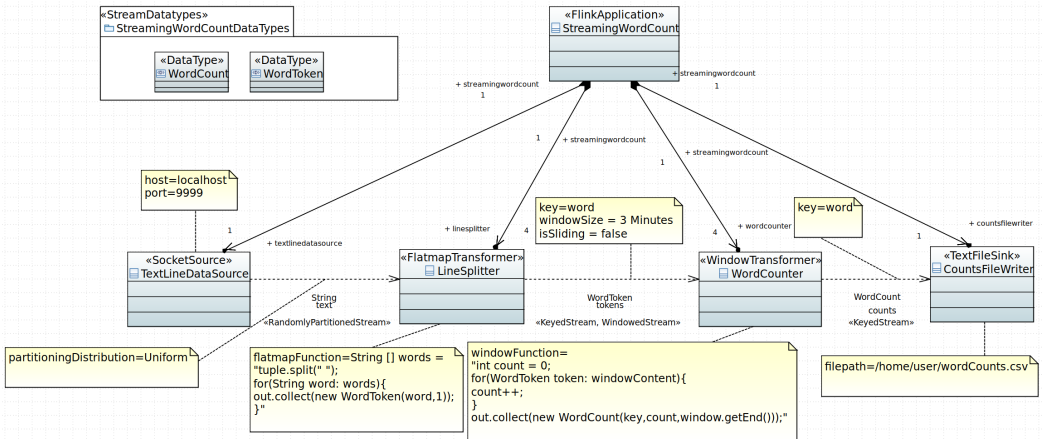


Fig. 3. Modeling the StreamingWordCount application using StreamUML: Class Diagram.

itself. For instance, *SessionWindows* are typically defined based on some inactivity *timeout*, i.e., a window is processed when no data has been received for a given amount of time.

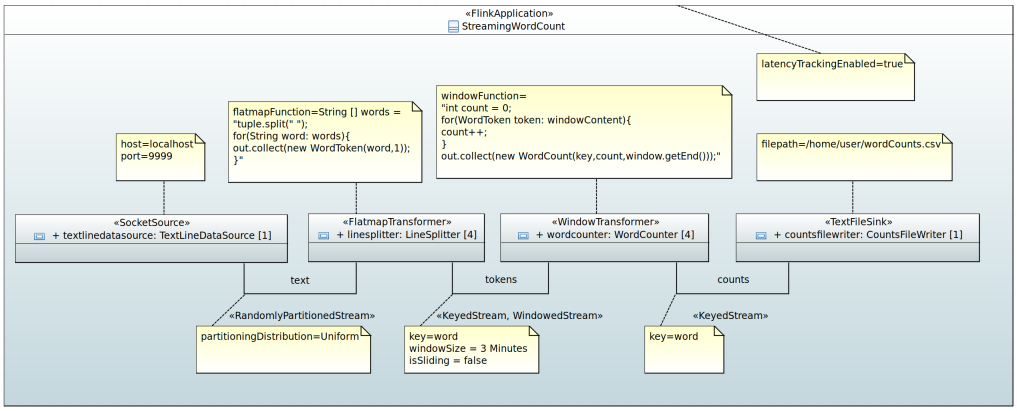


Fig. 4. Modeling the StreamingWordCount application using the provided UML profile: Composite Structure Diagram.

4.2 StreamUML: Technical Overview

StreamUML is defined as a UML profile. This choice has a number of benefits. First, UML is typically already used in design phases and known by developers. Moreover, tools enabling its adoption are available and offer proper extension mechanisms.

We also considered implementing our language as a textual DSL. However, given the declarative and visual nature of streaming applications, which can be represented as dataflow graphs, our hypothesis is that a graphical modeling approach provides greater benefits, as also demonstrated in [17].

We first analysed the UML specification to see what is already offered for modeling the flow of data between different components of an application. We found that UML includes the *InformationFlows* package [20], which "supports exchange of information between system entities at high levels of abstraction". InformationFlows are typically defined together with Class Diagrams, by using 1) *Information Flow*, represented as a dashed-line arrow (similarly to a UML Dependency) to model a unidirectional transmission of some information between two entities of a system (which can be modeled as instances of two UML Classes), and 2) *Information Item*, an extension of UML Classifier, to represent the information which is *conveyed* by an Information Flow element. According to the UML specification [20], UML Classes can be conveyed as well.

On the one side, InformationFlows provide fairly poor expressive capabilities, since they do not allow to model any detail about the nature of the conveyed information or the mechanisms by which it is transfered among different components. On the other side, they target the aspect of a software system that is of interest in streaming applications, i.e., the flow of data between different components.

Other candidate elements we have analysed are Activity Diagrams, which provide a notion of *object flows* [20] to model data transfers among Actions. Activity Diagrams are more known by UML users compared to InformationFlows and, besides the notion of object flows, offer also other features that are potentially useful in our case. However, a feasibility study concerning their usage in `StreamGen` (see Appendix A) has shown also some drawbacks, the main one being the fact that `StreamGen` users would have to define large models and to know many details about UML to be able to properly model their applications. As one

Stereotype	Type	Extends Stereotype	Extends Metaclass
«DistributedStreamingApplication»	Abstract	None	UML::StructuredClassifier
«StreamingOperator»	Abstract	None	UML::StructuredClassifier
«DataStream»	Abstract	None	UML::Connector
«StreamDataTypes»	Concrete	None	UML::Package
«FlinkApplication»	Concrete	«DistributedStreamingApplication»	None
«SparkApplication»	Concrete	«DistributedStreamingApplication»	None
«DataSource»	Abstract	«StreamingOperator»	None
«DataSink»	Abstract	«StreamingOperator»	None
«Transformer»	Abstract	«StreamingOperator»	None
«PartitionedStream»	Abstract	«DataStream»	None
«WindowedStream»	Concrete	«DataStream»	None
«KeyedStream»	Concrete	«PartitionedStream»	None
«RandomlyPartitionedStream»	Concrete	«PartitionedStream»	None
«BroadcastedStream»	Concrete	«PartitionedStream»	None
«NonParallelStream»	Concrete	«PartitionedStream»	None
«KafkaSource»	Concrete	«DataSource»	None
«SocketSource»	Concrete	«DataSource»	None
«TextFileSink»	Concrete	«DataSink»	None
«SocketSink»	Concrete	«DataSink»	None
«MapTransformer»	Concrete	«Transformer»	None
«FlatmapTransformer»	Concrete	«Transformer»	None
«SumTransformer»	Concrete	«Transformer»	None
«CountTransformer»	Concrete	«Transformer»	None
«ReduceTransformer»	Concrete	«Transformer»	None
«FilterTransformer»	Concrete	«Transformer»	None
«JoinTransformer»	Concrete	«Transformer»	None
«FlattenTransformer»	Concrete	«Transformer»	None
«NMapTransformer»	Concrete	«Transformer»	None
«NFlatmapTransformer»	Concrete	«Transformer»	None
«WindowTransformer»	Concrete	«Transformer»	None

Table 1. Some of the stereotypes provided by the StreamUML profile.

of the main goals of `StreamGen` is to streamline the prototyping of distributed streaming applications, simplicity and usability are key indicators.

We eventually identified the usage of Class Diagrams and their Information Flow package combined with Composite Structure Diagrams as the optimal solution, as it allows 1) to precisely model streaming applications according to the UML semantics and 2) to keep the modeling approach simple enough, satisfying the requirements we imposed for `StreamGen`. In this approach, Class Diagrams are used to define the application components and the exchange of information at an higher level of abstraction, while Composite Structure Diagrams are used to represents the application execution model, with instances of the previously defined components and data transfers modeled in more detail.

4.3 The StreamUML profile

Table 1 reports the most relevant stereotypes defined in StreamUML, which reflect the domain model classes defined in Section 4.1. For each stereotype, the table describes its type (second column), the generalized stereotypes (third column) and the extended UML meta-classes (fourth column). Figure 3 and Figure 4 show the StreamUML model for the StreamingWordCount application. The annotations in the model highlight, for the sake of clarity, the properties defined for each used stereotype. In the following we describe the model in detail, while describing the various aspects of StreamUML.

4.3.1 Defining the application topology. The functional decomposition typical of streaming applications is defined in terms of node types modeled as UML Classes. In the Class Diagram shown in Fig. 3, for instance, there are four classes that are then instantiated in the Composite Structure Diagram shown in Fig. 4. More specifically, the example shows a data source, two transformer nodes and a sink. Fig. 4 also highlights that the two transformers are available in four instances each. Data streams are modeled at the Class level through *UML Information Flow* elements, each of which must *convey* objects (i.e., tuples) of a specific type. At the instance level, i.e., in the Composite Structure Diagram, data streams are instantiated as properly stereotyped Connectors. In the example, there are three Connectors stereotyped as «RandomlyPartitionedStream», «KeyedStream, WindowedStream», and «KeyedStream».

4.3.2 Application data types. Instead of defining a dedicated stereotype to represent data types that are conveyed by the various data streams, we allow any UML DataType element to take the role of a data type, with the DataType attributes defining the various fields. In StreamUML, the data types conveyed in a streaming application are grouped in a UML Package stereotyped as «StreamDatatypes». In Fig. 3 we define `WordToken` and `WordCount` data types and associate them to the `tokens` and `counts` stream, respectively, while the `text` stream, i.e., the application input stream, conveys data of type `String` (already available in UML).

4.3.3 Streaming operators. StreamUML provides four *abstract* stereotypes to model the nodes of the dataflow graph representing a streaming application. These are the «StreamingOperator», «DataSource», «DataSink», «Transformer» stereotypes. All the provided *concrete* stereotypes extend these abstract ones. For instance, stereotypes are provided to represent various types of data sources, like the «KafkaSource» stereotype, which can be used to model a data source fetching data from Kafka, or the «SocketSource», which receives input data from a socket connection. In the StreamingWordCount running example, a «SocketSource» receives the input lines of text by listening on port 9999. Similarly, other types of data sinks are provided (e.g., text files, databases, etc). Again in the case of the StreamingWordCount application example, a simple «TextFileSink» is used to write the computed word counts into a specific text file.

4.3.4 StreamUML Transformers. The remaining stereotypes are aimed to model *transformers*. For some transformers the application designer does not need to go through any coding step, since their semantics is completely defined. For instance, the «SumTransformer» stereotype models an operator that, given a numerical field of the input data stream, continuously updates and outputs, as new tuples arrive, the cumulative sum of the field values. For some other transformers the designer flexibly defines the corresponding logic, since they only impose some constraints on the type of input and output streams. For example, the «MapTransformer» transforms each incoming tuple of the input data stream into *one and only one* tuple of the output data stream. The «WindowTransformer» needs as input a *windowed* data stream and transforms incoming *windows* of tuples into any number of tuples of the output data stream. The core logic of all these transformers needs to be specified by the designer. In particular, each of these stereotypes provides a specific property in which the designer directly writes the transformer logic in terms of snippets of Java code. In doing so, she can rely on the following variables:

- `tuple`: it allows to access the current tuple being processed and is available in functions that process one tuple at the time (e.g., «MapTransformer» and «FilterTransformer»).

- `key`: it allows to access the key associated to the current tuple being processed and is available only in those functions whose input stream is partitioned based on a given key (see later).
- `out`: it provides the `collect` method for emitting tuples in the output stream. It is available in transformers that allow to produce multiple tuples in their output data stream (e.g., «FlatmapTransformer» and «WindowTransformer»).
- `window`: it is a special object, available in a «WindowTransformer», allowing to access window metadata, such as its start and end time.
- `windowContent`: it is a Java iterator for accessing the tuples contained in the window being processed by a «WindowTransformer».

In the case of the StreamingWordCount application, a «FlatmapTransformer» (`LineSplitter`) takes as input the `text` data stream (which conveys the received lines of text), splits each line of text into separate words and, for each word, produces a new tuple in the output data stream (named `tokens`). The latter conveys tuples of type `WordToken` and is keyed and windowed as previously described. It is then processed by a «WindowTransformer», whose body simply counts the number of tuples within the current window being processed, and outputs a new tuple (of type `WordCount`) in the `wordCounts` stream. At code generation time, each snippet of code is statically checked to avoid the generation of code that would not compile. In particular, StreamCGM checks that each snippet of code is syntactically correct and that it only relies on locally defined variables along with those provided by the specific transformer (e.g., `tuple`, `key`, etc.). At the time of writing developers cannot rely on features such as auto-completion, syntax highlighting or other typical IDEs features while writing the custom transformer logic. We plan to develop support for such kinds of features in the future work.

4.3.5 Parallel execution. Each operator in the dataflow graph can be parallelized by the underlying execution framework, given the constraints defined at design time. The degree of parallelism is specified in the Composite Structure Diagram by setting the number of instances, through the `Multiplicity` property, of a given Class.. To allow parallel replicas to work on properly partitioned data, StreamUML supports various partitioning strategies, such as random partitioning (using different distributions), key-based partitioning and balanced partitioning. In the case of the WordCount application, `TextLineDataSource` and `CountsFileWriter` are not parallelized while both the `LineSplitter` and the `WordCounter` operators are parallelized with a factor of 4. While the first takes as input the `text` «RandomlyPartitionedStream» the second takes the `tokens` data stream, which is partitioned, using the «KeyedStream» stereotype, according to the value of the `word` field of each token (i.e., all tokens with the same `word` belong to same partitions).

4.3.6 Modeling stateful transformers. Some transformers can be *stateful*, meaning that they keep and manage an internal state, which is necessary given the kind of transformer being performed. An example is, again, the rolling sum transformer modeled through the «SumTransformer» stereotype. Under the hood the rolling sum transformer keeps as internal state the last produced value, so to produce an updated result upon arrival of the next tuple. In general, any kind of state can be internally kept (objects, maps, lists). StreamUML allows to model the internal state kept and used by *generic transformers*, e.g., «MapTransformer», «FlatmapTransformer», «WindowTransformer», etc. The internal state can be modeled by defining one or more UML Property elements for the Class representing the transformer. Such Property elements can be then accessed and modified within the body of the transformer.

```
1 self.base_InformationFlow.conveyed.oclIsKindOf(UML::PrimitiveType) = Bag{true} or
2 (self.base_InformationFlow.conveyed.owner.getAppliedStereotypes().name -> includes('
    ↪ StreamDatatypes') and
3 self.base_InformationFlow.conveyed.oclIsTypeOf(UML::DataType) = Bag{true})
```

Listing 4. Example OCL constraint checking the validity of the conveyed data types.

4.3.7 Windowing policies. By using the «WindowedStream» stereotype, one can set a window on a specific data stream. The stereotype allows to specify the various windowing policies described in Section 4.1. In particular, time-based windows and session windows are supported. In the case of the StreamingWordCount running example, a *fixed, time-based* window of 3 minutes has been specified on the `tokens` data stream.

4.3.8 Modeling platform-specific settings. According to our experience, some platforms require the definition of platform-specific yet application-dependent settings. For instance, since Spark is based on the micro-batch processing approach [29], it requires the *micro-batch size* to be defined. This parameter comes from the way Spark processes data, but is, in general, not necessary to specify the application logic of a streaming application. Still, when adopting Spark, the designer might be willing to tune such parameter, since it has an impact on the application performance. StreamUML provides a dedicated stereotype for each supported platform to allow the configuration of the various platform-specific parameters. Examples are the «SparkApplication» stereotype providing, for instance, the *microBatchSize* property, or the «FlinkApplication» stereotype, which provides the *latencyTrackingEnabled* property (see Figure 4). These additional stereotypes extend the UML Model meta-class and are to be applied on UML Models containing streaming applications, in order to specify and configure the target platform. A Model should specify at most one target platform by owning at most one of such platform-specific stereotypes. This constraint is due to the fact that StreamCGM looks at this stereotype to know the target platform for which code has to be generated. This currently comes with the drawback of not allowing multiple configurations for different platforms to be simultaneously defined on a given model. We plan to relax this constraint in the future work by directly specifying the target platform in the launch configuration of StreamCGM. This will further simplify the exploration of multiple target alternatives.

4.3.9 Consistency Checking of StreamUML Models. We enriched StreamUML with a number of constraints, to be set in the Class Diagram, to assist designers in creating valid models. Such constraints have been specified directly within StreamUML using OCL and, thus, can be interactively checked while creating the model. Constraints also make sure that a model can be translated into code without errors. As an example, the constraint reported in listing 4 is applied on the «DataStream» and checks for the validity of the various conveyed data types, which must be either primitive types or must be DataTypes contained in a «StreamDatatypes».

Other constraints impose different modeling rules, e.g., data sources should not have an input stream and should have a single output stream (and vice versa for data sinks), the input stream of a «WindowTransformer» must be a «WindowStream». Moreover, we defined a number of platform-specific OCL constraints to validate a model against specific restrictions imposed by the target execution platform. For instance, back to the micro-batch size needed in Spark, the constraint reported in listing 5 checks that all the window sizes of a Spark applications are a multiple of the micro-batch one.

```
UML::InformationFlow.allInstances()
->select( i| i.getAppliedStereotypes() ->asSequence()
-> exists(s | s.name = 'WindowedStream')
)
->collect(w | w.getValue(w.getAppliedStereotypes()
-> select(e | e.name = 'WindowedStream')
-> asSequence() ->first(), 'window').eGet('size').eGet('size'))
->forAll(s | s.oclAsType(Integer).mod(self.getValue(self.getAppliedStereotypes()
-> select(e | e.name = 'SparkApplication')->asSequence() ->first(), 'microBatchSize'))
    ↪  = 0)
```

Listing 5. OCL constraint checking the windows used in a Spark application have all a size which is a multiple of the micro-batch size.

It is worth to note that in our OCL constraints we make use of the *getAppliedStereotypes()* primitive, which is not standard, but specific of the UML2 implementation in Eclipse. If, on the one hand, this limits the usability of StreamUML within the context of a specific platform (the one on top of which the `StreamGen` tool has been developed), on the other hand, it provides us with a practical solution to predicate on stereotyped UML elements within OCL constraints. Moreover, other platforms may provide different means to access stereotypes attached to modeling elements, which would allow the definition of equivalent constraints. In the worst case, a subset of our OCL constraints would not be available in implementations of StreamUML on platforms different than Eclipse.

5 CODE GENERATION

In this Section we focus on StreamCGM, i.e., the code generation module of `StreamGen`. StreamCGM is based on templates technology and has been implemented in Acceleo. Both the Spark and Flink code generators included in StreamCGM produce Java code. This decision was due to the fact that most of the streaming platforms we analyzed provide Java APIs. Some of them do actually provide APIs implementations also for other languages, such as Scala or Python, but this is a less common case and often such implementations are still immature and unstable.

StreamCGM takes as input StreamUML models and starts looking for the stereotype that identifies the target platform of choice (i.e., the «SparkApplication» and the «FlinkApplication» stereotypes). Based on this, it activates the corresponding code generator, which starts by creating a Maven project already configured and linked to all the required target platform dependencies. If specific types of data sources/sinks are used (e.g., a «KafkaSource» or a «CassandraSink»), a Maven dependency to import the corresponding driver is added. Then, the generated code is structured into three packages: one for the data types used by the application, one for the user-defined functions and one for the application builder and launcher. As a consequence, the code generation features three phases:

(1) **Data types generation**: in this phase the code generator creates a new package that is populated with the necessary classes for all the *data types* used in the model. In particular, the generation looks for UML Packages tagged with the «StreamDatatypes» stereotype and, for each nested data type, creates a corresponding Plain Old Java Object (POJO). This phase is independent of the selected target platform.
(2) **Streaming functions generation**: in this phase the code generator creates a new package that is populated with the necessary classes implementing the various user-defined operators present in the model. In particular, the code generator looks for <<StructuredClassifier>> meta-typed entities tagged with a stereotype that requires

```java
public class StreamingWordCount {

  private static final Logger logger = LoggerFactory.getLogger(StreamingWordCount.
  ↪ class);

  public static void main(String[] args) throws Exception {

    final StreamExecutionEnvironment env = StreamExecutionEnvironment.
  ↪ getExecutionEnvironment();

    DataStream<String> text = env
        .socketTextStream("localhost", 9999);

    DataStream<WordToken> tokens = text
        .flatMap(new LineSplitter())
        .setParallelism(4);

    DataStream<WordCount> counts = tokens
        .keyBy("word")
        .timeWindow(Time.minutes(3))
        .apply(new WordCounter())
        .setParallelism(4);

    counts
      .keyBy("word")
      .writeAsText("/home/user/wordCounts.csv")
      .setParallelism(1);

    JobExecutionResult result = env.execute();
    logger.info("EXECUTION TIME: " + result.getNetRuntime(TimeUnit.SECONDS));

  }
}
```

Listing 6. The generated Java class that assembles the dataflow graph for our running example.

the user to specify the transformer logic (e.g., «MapTransformer», «WindowTransformer», etc.) and, for each of them, creates a new Java class that implements specific interfaces provided by the selected target platform.

(3) **Streaming job generation**: in this phase the code generator creates a new package that contains the actual application to be executed, i.e., a Java class that builds up and executes the dataflow graph defined in the model. For instance, Listing 6 shows an extract of the Flink code generated from the StreamingWordCount application model, which takes care of assembling and executing the dataflow graph. In this Class, the various *functions* created in the second phase (i.e., `LineSplitter`, `WordCounter`) are instantiated and connected using data streams (i.e., `text`, `tokens`, `counts`) that transport tuples of the various *data types* defined in the first phase (i.e., `WordToken`, `WordCount`).

Although the generation workflow is the same, the generators targeting different platforms work differently (with the exception of the first phase). For instance, referring to the «SumTransformer», Flink provides explicit APIs for computing a cumulative sum over a numeric field of a data stream, while Spark does not. As a consequence, while the Flink generator only has to output a single line of code, the Spark generator needs to add lower level code, which, in our current implementation, results in 20+ lines of code. Another example is the case of data streams windowed using a fixed time window. Flink explicitly allows to set a fixed time window on a data stream, while in Spark this can be obtained as a sub-case of the sliding time window, in which the length of the slide is equal to the size of the window itself.

6 EVALUATION

In the evaluation of `StreamGen` we ran a mixed-method experimental champaign with the goal of evaluating various aspects of `StreamGen` connected to the satisfaction of the requirements stated in Section 3. In particular we performed the following set of experiments:

- In order to experiment with the creation of an application complex enough to exploit most of the modeling elements provided by StreamUML, we have defined our own case study, called *TempTrack*, for temperature tracking that combines 11 different operators and 10 streams and integrates both Kafka and Cassandra, thus showing a good level of complexity (Section 6.1).
- In order to demonstrate `StreamGen` ability to support modeling of real streaming applications and to generate their code for different streaming platforms with performance comparable to manually developed code, we have modeled and generated the code for the Yahoo streaming benchmark [10], which includes pre-existing implementations of the same application for both Flink and Spark. This experiment shows, on the one hand, the ability of `StreamGen` to deal with real-world applications, on the other hand, that the performance of the generated code is very close to the one offered by the manual implementations (Section 6.2).
- In order to assess the completeness of StreamUML, we have compared its modeling capabilities, i.e., the provided streaming concepts and features, against the Google Dataflow Model. The latter is considered a state-of-the-art model for stream processing systems. We show that `StreamGen` is able to support most of the proposed modeling elements (Section 6.3).
- In order to assess the usability of `StreamGen` and the extent to which it enables fast prototyping even when technical expertise of developers is low, we ran a user-study [26] featuring 23 randomly-sampled and unexperienced users (Section 6.4).
- In order to provide an assessment of `StreamGen` extensibility, we ran an internal experiment in which the first author extended the tool adding support for Apache Apex while systematically reporting on the needed steps and their complexity (Section 6.5).

6.1 Case Study: Processing Sensors Data

To show its practical applicability, we have used `StreamGen` to develop a case study application, that we called *TempTrack*, which takes as input and processes the temperatures measured by various sensors spread over the rooms of a set of buildings. The StreamGen Composite Structure Diagram for this example is shown in Figure 5. Details are omitted for the sake of space, but the complete model and generated code are available online [11]. *TempTrack* performs the following steps:

- Data is received from temperature sensors (through the `TemperatureSensor` component in the model) and passed to the `TemperatureParser` operation which wraps them into objects of type `RoomTemperature` (see the conveyed type of the `parsedTemperatures` stream).
- Room temperatures are passed to `ClearRawData` for cleaning and then stored into a Cassandra [12] database for future analysis.
- Room temperatures are also streamed toward `ComputeRoomStatistics` that computes the average and the maximum temperature of each room over time windows of 2

[10] https://github.com/yahoo/streaming-benchmarks
[11] https://github.com/MicheleGuerriero/streamgen/tree/master/temptrack
[12] http://cassandra.apache.org/

minutes. The results are passed to a sink that creates an external .csv file available for further analyses.
- The statistics are also streamed into `MonitorCriticalPrediction` that continuously predicts the average temperature of each room in the next 10 minutes and streams an alert into a Kafka [13] queue if the predicted value exceeds a given threshold.
- Finally, the same statistics are also streamed into `CheckCriticalTemperature` that, in turn, streams an alert into a Kafka queue if the average temperature of a room during the last 3 seconds exceeds a given threshold.

TempTrack integrates with two external components (Kafka and Cassandra) and features various different streaming concepts and a total of 11 operators. Moreover, it shows quite strict real-time constraints for the detection and prediction of critical events and a high sensing rate for the sensors. As such, it can be considered a non-trivial example suitable to test the expressiveness of StreamUML and the effectiveness of StreamCGM generation. From the above model, which is suitable for describing the features of the application, we have been able to generate and execute the code for both Spark and Flink (code available at the same link of the model). Moreover, by measuring the latency associated to both implementations of *TempTrack*, we could confirm that the code generated for Flink better fits our requirements, as it was actually expected given the micro-batch nature of Spark that is not the best choice for latency minimization.

6.2 Modeling the Yahoo Streaming Benchmark

The Yahoo streaming benchmark has been developed to compare the performance of different streaming platforms. It consists of an application for analyzing advertising events (*ad events*). The application receives ad events (e.g., views, click, etc.) through a Kafka instance. Each ad event is parsed, filtered, cleaned and enriched with the id of the advertising campaign it belongs to. Such correspondence is retrieved from a Redis database. Then, ad events are grouped by campaign and by time windows of 10 seconds. Finally, the count of ad events for each campaign and for each window is computed and stored back into the Redis database. Benchmark implementations are available for Storm, Spark, Flink and Apex and allow us to gather and compare performance metrics, specifically, computation latency.

We modeled the benchmark with `StreamGen` to show that it offers at least those design elements that are required by this third-party application. Moreover, we generated the corresponding code for the two platforms we support and we compared the achieved result with the one originally developed by Yahoo, both in terms of code quality and performance.

Figure 6 shows the model we designed (some comments reporting stereotypes configuration have been omitted for the sake of space). Indeed, the benchmark relies on streaming features that are currently all supported by `StreamGen`, such as windowing, key-based partitioning (i.e., ad events are grouped by advertising campaign), flatmap and filtering operations, as well as stateful computations. `StreamGen` also supports Kafka as a data source. The only limitation we found was merely technological, as `StreamGen` does not currently support Redis as a data source/sink. To overcome this limitation, in our experiment, the application results are stored back into a second Kafka queue (see the `CampaignSink` «KafkaSink»). Moreover, the mapping between ads and campaigns is fetched at the beginning of the execution from a text file and is then kept in memory by the `CampaignJoin` «NFlatmapTranformer», which is responsible for matching events with campaigns. This allowed us to test `StreamGen` support

[13] https://kafka.apache.org/

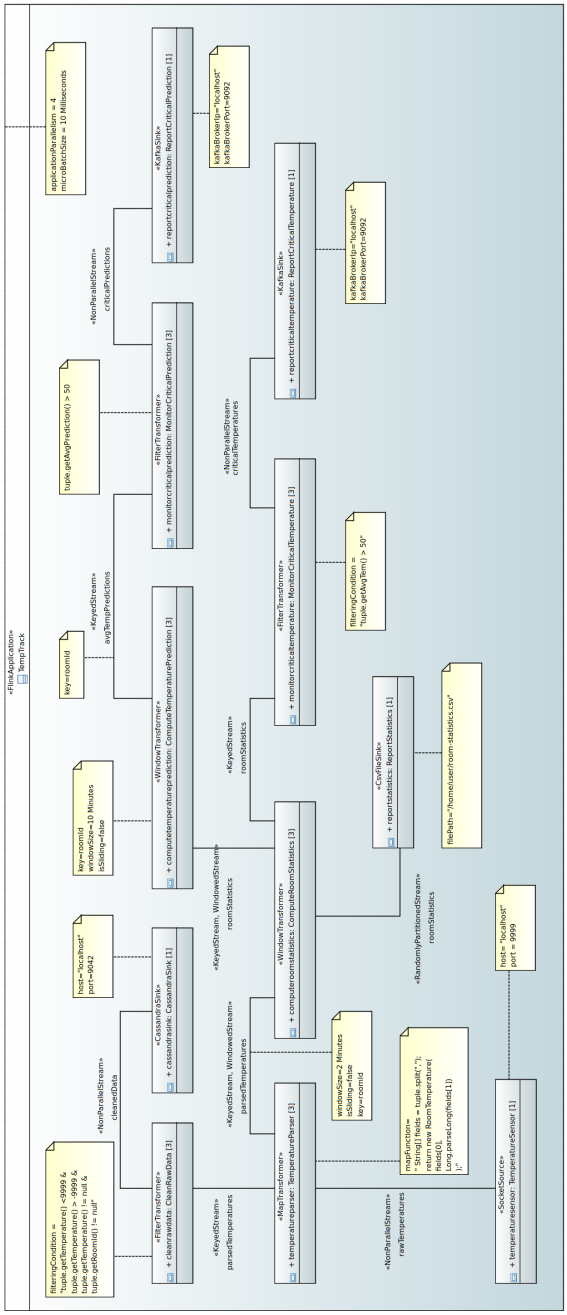


Fig. 5. StreamUML model for the *TempTrack* application case study.

for stateful transformers, with the mapping between ads and campaigns being the internal state. The model along with the generated code are available online [14].

[14]https://github.com/MicheleGuerriero/streamgen/tree/master/yahoo-benchmark

We generated the code both for Flink and Spark and compared it with the original implementations. As for Flink, the code structure of our generated implementation resembles the original one and shows only 9% more lines of code (see the first row, first two columns of Table 2). In fact, as expectable from a model-driven development tool, `StreamGen` generates some boilerplate code. For example, it does not concatenate multiple transformers to produce a single resulting data stream, but generates specific code for each InformationFlow element, even for those that are just intermediate (and never actually used) streams along a chain of transformers. Another example are the `toString()`, `hashCode()` and `equals()` methods, which are always overridden when generating data types. As for Spark, we could not compare the code because the original implementation is written in Scala while `StreamGen` generates Java code.

The performance metric provided by the Yahoo benchmark is the latency of computation for each campaign window that is produced, i.e., the time *"from when the last event was emitted to Kafka for that particular campaign window and when it was written into Redis"*. For our experiments, we instrumented the applications generated by `StreamGen` in order to measure the same metric, with Redis being replaced by Kafka. We also made sure to use the same input data used in the benchmark. In particular, the Yahoo benchmark randomly generates the workload: first it generates the mapping between ads and campaigns and stores this mapping into Redis; then it starts loading ad events into Kafka. In our experiment we used the same load generator, which has been adjusted in order not to write the mapping between ads and campaign into Redis, but rather into a CSV file, which is then read by the generated application (see the `AdToCampaignSource` «TextFileSource» in Figure 6). We ran both the original Yahoo benchmark as well as the `StreamGen` generated versions (both for Flink and Spark) on an Amazon EC2 m5d.xlarge instance, which comes with 16 GB of RAM and 4 CPU cores. In every experiment, we set the application parallelism to 4, i.e., each operator had 4 replicas and we computed the average latency of computation across all the campaign windows that were produced. Such average is further averaged over ten runs of each application. The second row of Table 2 reports the results of these experiments. The application generated by `StreamGen` performs in general worst, both in the case of Spark and Flink, with the average latency being increased by 4.78% in the case of Flink and 7.21% in the case of Spark, which is acceptable although non negligible. A possible explanation for this can be found in the data structures that are internally used by the various operators. In particular, in the applications generated by `StreamGen`, the *CampaignJoin* operator stores the mapping between ads and campaigns as a list of objects of type `AdCampaign`. When a new ad event arrives, the operator performs a linear search into this list in order to find the corresponding campaign. Instead, in the original Yahoo benchmark, the mapping is stored as a hash map with constant access time. The reason why the generated applications use a list is that the current implementation of `StreamGen` does not provide a way to model an hash map as the internal state of a stateful operator. Extending `StreamGen` in this direction could make performance closer to that of the original benchmark. We plan to further validate such hypothesis in future work.

6.3 Evaluating StreamGen Feature Offer against Google Dataflow

The Google Dataflow Model [2] was proposed in 2015 by Akidau et al. as a reference model for the capabilities a modern stream processor should offer. The main idea behind the Google Dataflow Model is to bridge the gap between the massive-scale processing requirements imposed by modern streaming data sources and the sophisticated functional requirements that streaming application providers have identified, such as the ability to work with different

	Flink Orig	FlinkStreamGen	SparkOrig	Spark StreamGen
LOC	608	666	n.a.	n.a.
Latency (ms)	9957.84	10433.82	9984.83	10704.73

Table 2. Analysis of `StreamGen` generated implementation of the Yahoo benchmark.

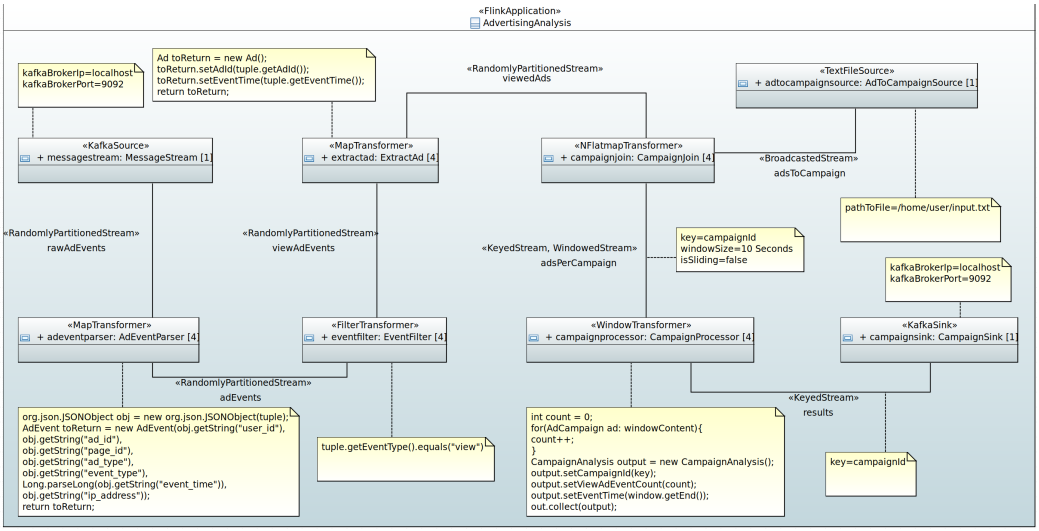


Fig. 6. StreamUML model for the Yahoo streaming benchmark.

windowing and time semantics or to trade correctness of results, latency of computation and cost when processing large-scale data streams. The model addresses various aspects that are summarized in Table 3 and compared against what `StreamGen` offers.

As for types of data processing operations, `StreamGen` covers almost all the identified aspects except (1) the ability to define *composite transforms*, i.e., custom transformers obtained by composing the primitive ones that are available out-of-the-box and (2) the ability to specify *metrics* to be gathered at runtime. Regarding methods for splitting data streams, `StreamGen` supports modeling of all the possible windowing semantics defined by the Google Dataflow Model, but, at the moment, it does not allow to specify custom (i.e., user-defined) windowing policies. As for the mechanisms to start window processing, `StreamGen` does not allow designers to specify custom triggers to decide when windows should be considered ready to be processed. For the moment it only supports one triggering semantics that closes a window when the specified amount of time has passed. Finally, `StreamGen` lacks the ability to define how refinements of previous results are handled. For instance, with the *accumulation* policy, window contents are left intact in memory and later tuples cause the window to be re-processed and results to be re-computed. The default semantics supported by `StreamGen` at the moment is *discarding*, i.e., once results are emitted the window content is discarded and later results have no relation with the previous ones.

Overall, we conclude that `StreamGen` does currently support various features defined by the Dataflow Model, but also has several limitations. We do not envision any conceptual

Category	Dataflow Model Feature	In StreamGen	How it can be modeled
Types of data processing operations	ParDo	Yes	FlatmapTransformer
	GroupByKey	Yes	KeyedStream
	Flatten	Yes	FlattenTransformer
	Combine	Yes	JoinTransformer
	CompositeTransform	No	n.a.
	Side Inputs	Yes	Transformer with multiple input streams
	Source API	Yes	DataSource substereotypes
	Metrics	No	n.a.
	Stateful processing	Yes	Sate modeled as attributes of an operator.
Splitting data streams	Global windows	Yes	WindowedStream + NonParallelStream
	Fixed windows	Yes	Modeled through the WindowedStream
	Sliding windows	Yes	Modeled through the WindowedStream
	Session windows	Yes	Modeled through the WindowedStream
	Custom windows	No	n.a
Starting window processing	Triggers	No	(default processing time trigger)
Handling refinements of previous results	Accumulation	No	n.a.
	Discarding	No	n.a.
	Accumulation and retraction	No	n.a. (currently supported by no stream processor)

Table 3. Benchmarking `StreamGen` against Google Dataflow.

issue that would prevent `StreamGen` from completely covering the Google Dataflow Model. Most of the currently lacking features, such as the ability to specify how refinements of previous results are handled or which metrics have to be gathered at runtime, could be specified as configurations for the various streaming operators. Other features, such as the ability to define composite transforms could require the definition of some new stereotypes. In any case we expect the foundation of the overall modeling approach to remain the same. We plan to verify such hypothesis as future work.

6.4 `StreamGen`: User-Study Results

To understand the extent to which users are able to exploit `StreamGen` to develop a streaming application, we have run a user-study featuring 23 randomly-sampled practitioners with *no* previous knowledge over Big Data applications and having no programming skills. The population in question featured: (a) a majority of female practitioners, with a 47% presence of male; (b) an age range between 20 and 35 (with 86% between 20 and 25); (c) an almost equal geographical and cultural extraction distribution across Europe, with about 33% majority of Italian participants; (d) educational background diversity equally arranged in scientific degree (no computer scientists), engineering degree as well as economics.

The population was asked: (1) to modify a streaming application for processing financial transactions and (2) to fully develop an application for processing data coming from temperature sensors. The user-study protocol was structured as follows. First, a 3-hour lecture was used to illustrate the basic concepts of stream processing and distributed streaming applications. Then, users were provided with a virtual machine instrumented with

very simple	simple	medium	complex	very complex
0%	47.8%	34.8%	4.3%	13%

Table 4. Distribution of the perceived level of difficulty (Likert Scale) of the proposed application to be developed.

a pre-configured `StreamGen` installation. There was no specific training on the usage of the tool, but all study participants received a user manual describing its main features.

Subsequently, the population was given with the predefined model of an application that had to process incoming financial transactions and compute simple statistics (e.g., average expenses per consumer, number of consumers who spent more than a given amount of money during the last hour, etc) and was asked to generate the code, modify the application (e.g., changing the parallelism of some operators and the partitioning of some data streams, as well as computing new statistics) and re-generate the code. Then, the population was asked to proceed on a more complex exercise asking to develop from scratch an application that processes sensor-monitored temperature data points over a set of rooms, to identify criticalities. Overall, the study participants were given 2 hours to copy and run the virtual machine on their computer, get accustomed to `StreamGen`, and work on the two examples. Finally, the population was surveyed to understand the extent to which `StreamGen` achieves its hypothesised goal of increasing productivity in the development of distributed streaming applications. The survey and answers are available online [15]. We focused on a single key research question, namely, *"to what extent does StreamGen aid in simplifying the understanding, development, and operation of distributed streaming applications?"*.

First, Table 4 reports the distribution of the perceived level of complexity of the two proposed applications. On average the applications were considered of *medium* complexity. At the end of the two hours, one participant could not complete any of the exercises, as she had troubles installing the VM due to her system configuration, while 37% of the users completely developed and ran the proposed applications. The remaining 63% declared they partially completed both exercises with an overwhelming majority of 96% that completed the first exercise successfully. Of such 96%, 70% declared to have not finished the second exercise because of time limitations. Furthermore, among those who managed to completely develop at least one application, 45% declared that they spent around half an hour to design their model, generate, and run the code, while the remaining 55% declared they spent around 1 hour. Finally, Table 5 reports the extent to which our users found `StreamGen` useful towards understanding and learning how to build distributed streaming applications as opposed to digging into the code of a specific streaming framework. Around 74% found `StreamGen` to be *useful* or *very-useful*. Conversely, we asked the candidates whether they found issues during their exercises. About 30% of the practitioners reported between 1 and 3 issues during their experimentation; for example, one practitioner remarks: *"[for Ex. 2] I faced some issues with the code generation [...] because of configuration problems"*.

User-Study Summary. The overwhelming majority of users highlight the positive role of the proposed `StreamGen` solution in understanding and developing distributed streaming applications. However, a statistically relevant proportion of participants (30%) reports difficulties in code-generation which still need attention.

[15] https://tinyurl.com/yd8am8yd

not useful	of little use	neutral	useful	very useful
0%	4.3%	21.7%	34.8%	39.1%

Table 5. Distribution of the results about how much (Likert Scale) StreamUML simplifies the understanding of distributed streaming applications.

6.5 Preliminary Assessment of `StreamGen` Extensibility

`StreamGen` comes as a solution to deal with the technology heterogeneity in the field of distributed streaming frameworks. The main idea is to let `StreamGen` users to quickly familiarize with streaming applications through `StreamUML` and to easily and seamlessly experiment with multiple streaming frameworks. Because of this, `StreamGen` is somehow expected to be extensible, in such a way that new streaming frameworks can be easily and independently plugged in. In this Section we report our own experience extending `StreamGen` to support Apache Apex [16]. We outline the steps that are needed in order to add support in `StreamGen` for additional streaming frameworks, showing that the effort in extending `StreamGen` is reduced thanks to the provided abstractions and to the way it has been engineered.

Obviously, the first mandatory step for one wishing to extend `StreamGen` is to learn and acquire substantial experience with the framework being added. This is quite obvious given that, if we want to release `StreamGen` users from the burden of knowing multiple frameworks, someone have to do the hard work to properly map framework specific concepts to `StreamUML` abstractions.

After this preliminary step, one can begin the actual extension of `StreamGen`. In terms of extensibility, the main advantages of the `StreamGen` approach are (1) the decoupling between the language used for defining an application and its concrete implementation and (2) the modular architecture of `StreamCGM`. On the one hand, the only change that needs to be done on `StreamUML` is the addition of one more stereotype extending the «DistributedStreamingApplication» one, to allow the streaming framework being added to be set as target platform for a given application model. For instance, in the case of Apex, we added the «ApexApplication» stereotype. On the other hand, `StreamCGM` needs an additional module that encapsulates the code generation logic for the streaming framework being added. Nevertheless, the generation of data types is expected to be the same for every streaming framework, since data types are really just POJOs.

As a consequence, adding such code generation module requires the development of:

- the generative logic for the various types of sources, sinks and transformers;
- the generative logic for assembling the application dataflow;
- the generative logic for the pom.xml file needed for managing the generated project.

According to our experience in adding support for Apex (and actually the same also happened when adding support for Spark), the most complex step is the definition of the generative logic for the various types of transformers. The main obstacles are the different ways in which different frameworks natively support streaming operations (e.g., windowing) and higher level operators (e.g., sum, count, filter, etc.). Indeed, when native support is provided, the generative logic is quite straightforward as it consists, in most of the cases, in properly invoking functions. Conversely, when no native support is available, like in the case of

[16] https://apex.apache.org/

«WindowTransformer» or of «SumTransformer» in Spark, the generative logic becomes more complex, since it must generate an implementation of such operators that uses lower level APIs. For instance, in order to generate «WindowTransformer» for Spark, a windowed *reduce* function is applied on the input stream in order to aggregate the incoming tuples into windows. Then they are processed by a *flatmap* function, which encapsulates the actual logic for processing windows.

We conclude that extending `StreamGen` mainly means adding a code generation module to `StreamCGM` and that this can be done in a quite systematic way. Moreover, the needed effort for developing such module strictly depends on the native streaming capabilities of the framework being added. It is also worth noting that `StreamCGM` refers to specific versions of the supported target platforms. In particular, currently `StreamCGM` generates code for Flink 1.4.0 and Spark 2.3.0. Updating the version of the generated code in the future could imply the need for updating the corresponding code generation module, in the case some APIs have changed.

6.6 Summary of the experiments

Given the results of our experiments, we conclude that `StreamGen` reasonably fulfills its requirements, even if, as we have highlighted in the previous sections, there is space for further research and consolidation of the current prototype. First, we were able to use it to model and experiment with a relatively complex application case study, which exploits multiple streaming features, is composed of 11 independent streaming operators and relies on 2 external systems (namely, Kafka and Cassandra). Second, we showed that `StreamGen` generates Flink and Spark code for the Yahoo Benchmark application introducing only a small performance overhead. Third, we have compared `StreamGen` linguistic constructs against the Google Data Model and concluded that it is able to cover most of the features defined in this model. Fourth, our preliminary empirical study shows that around half of our users were able to prototype the example streaming application in less than one hour on average. This preliminary result can be considered significant both in terms of increased productivity and learning curve smoothing, even if it needs to be confirmed by other studies involving streaming experts and real industrial scenarios. Finally, we have experimented with the extensibility of `StreamGen` and we have shown that it is a relatively simple task for an expert developer.

7 THREATS TO VALIDITY

Due to the nature of the evaluation campaign behind `StreamGen`, we elaborated threats to validity according to typical evaluation schemes adopted for case- and user-study experimentation, most notably, the framework by Feldt et al. [12].

Transferability validity. Our extensibility analysis has been carried out by the developer of the tool itself, so it may be biased by his judgment. A possible mitigation strategy [12] would be to repeat the experiment, i.e., developing support for other streaming platforms, but relying on some external developers. This would help in better assessing not only the steps that are needed to extend `StreamGen` but also its complexity and overall time to add support for a new technology.

Credibility and Dependability validity. The TempTrack case study suffers from being an internal case study that we intentionally developed with the goal of stressing most of the `StreamGen` provided features. As a way to mitigate possible biases, we complemented this experiment with the comparison against the Google Dataflow Model and with the adoption

of the Yahoo Benchmark. Of course, more examples would be needed to completely overcome the threat. We plan to focus on this aspect in future work.

As for our experiment with the Yahoo Streaming Benchmark, one threat is that it can be considered as a pretty old benchmark, being the last update from May 2017. Nevertheless, it is still considered as the reference benchmark when it comes to streaming applications. Moreover, to the best of our knowledge, there is currently no newer or more reliable streaming benchmark available.

One may also argue that we are supporting only two streaming platforms, which is quite a small number to claim `StreamGen` deals with the technology heterogeneity. Nevertheless, we are currently finalizing the Apex support discussed in Section 6.5, for which we do not envision any technical issue. Additionally, it is worth to remark that Flink and Spark are currently among the most used and supported streaming platforms, along with being the most comprehensive in terms of provided streaming features. Of course the ideal mitigation strategy for this threat would be to just add support for all the available platforms.

Confirmability validity. Finally, the user study we conducted to offer a user-based evaluation of `StreamGen` itself constitutes an example evaluation which, although with limited generalisability power, offers an overview of how `StreamGen` can be used in a real-life context and how users react to its usage. Although we adopted an appropriate sampling approach (and control factors therein) to warrant for a rigorous user-study, we cannot claim generalisability beyond populations consistent with our sample. Further user-studies and case-studies are planned to strengthen this limitation.

8 RELATED WORK

Model-Driven Engineering (MDE) has been applied in several application contexts to build various types of applications. For instance, Balasubramanian et al. proposed a model-driven development approach for distributed embedded automotive applications [6], Almeida provided a methodology for the model-driven design of distributed applications [3], MODA-Clouds [5] is an integrated solution for the model-driven design and operation of multi-Cloud applications.

MDE has been already applied in the context of Big Data. Authors of [23] focus on modeling and generating code for Hadoop MapReduce. In the streaming world, a similar support is offered by Stormgen [27], a DSL for Storm-based *topologies*. With respect to these efforts, we strive to provide a higher level modeling language that abstracts from a specific framework. Moreover, we implemented this language as a UML profile, with all the advantages coming from relying on a well-known, standard and widely-used modeling notation. Another MDE approach targeting specifically streaming applications is Juniper [10]. It focuses on real-time systems running on HPC and FPGA platforms, while, to the best of our knowledge, does not target distributed streaming platforms.

A recent effort towards using UML for modeling big data applications is offered by DICE [8, 13]. In this case, however, the emphasis is on quality analysis at the modeling level and on generating infrastructural code suitable for automatic deployment. Instead, application-level code generation is not considered. For this reason, we see `StreamGen` and DICE as complementary, although their integration deserves further study and engineering.

TOREADOR is a research project which, similarly to us, addresses the problem of fostering the wide adoption of Big Data solutions by means of a MDE approach [4, 11]. TOREADOR relies on a predefined model of a big data application (similar to a template) to be customized following the steps of the proposed methodology. TOREADOR also offers some degree of automation to generate and deploy the DIA. With respect to `StreamGen`, the focus of

TOREADOR is more on architectural/structural aspects of the application and on the steps to be followed in the design phase. For example it allows to deal with non-functional requirements (e.g., performance, data protection) and to specify deployment choices (e.g., the target Cloud provider for deploying the application). At the same time, it does not offer much in terms of modeling the application logic. `StreamGen`, instead, allows the developer to integrate the application logic in the design model and is not constrained by the adoption of a predefined model for the application architecture. Moreover, TOREADOR is deployed as a service and is not based on UML, while `StreamGen` is integrated in a widely used and general purpose IDE and, as one of its contributions, it builds on top of UML.

Apache Beam [17] is similar to `StreamGen` as it allows streaming applications to be executed on multiple frameworks, but it does so by adopting a completely different approach. Beam is an implementation of the Google Dataflow Model [2]. It offers a set of programming abstractions to design data pipelines, an API that implements the abstractions, and a set of *runners*. Each runner executes Beam pipelines on a specific execution engines (e.g., Spark, Flink, Gearpump). While `StreamGen` provides a modeling approach based on UML, Beam provides its own APIs, which are, at least to our experience, non trivial and require quite good programming expertise. Indeed, the goal of `StreamGen` is to smoothen the process of entering the world of streaming applications and to simplify and speed up their adoption and we believe modeling can be key along this direction. Moreover `StreamGen` works by pure code generation while Beam runners do not generate code but rather *interpret* Beam-coded pipelines. Thus `StreamGen` still offers the possibility to customize the generated code, for instance if one wants to exploits some specific APIs provided by the target platform, something that is currently not possible with Beam.

Summing up, compared to the related literature, `StreamGen` shows the following unique aspects: being based on a high-level, UML-based language, it aims at smoothen the process of learning how to develop a streaming application; it allows developers to define both the structure of the streaming application, the communication logic between its components, and the application logic; finally, it offers a code generation mechanism that enables an easy adoption of an extensible set of distributed streaming frameworks, with an overhead which has been shown to be small in the considered cases.

9 CONCLUSION AND FUTURE WORK

The increasing demand for distributed streaming applications creates the need for appropriate abstractions and connected languages and tools to cope with their complexity and heterogeneity. Along this direction, we outline `StreamGen` a model-driven development tool composed of a novel UML profile, which allows to define the application model using the familiar UML notation, and of two code generators for Apache Spark and Flink. By evaluating `StreamGen` we have shown that it can be used to design and develop real world applications (i.e., the Yahoo benchmark) and that, despite some current limitations, it provides support for many relevant streaming features. Moreover, our user study shows that `StreamGen` does indeed provide practical impact, as it allows fast prototyping of streaming applications also to non-expert users. Finally, we discussed how `StreamGen` can be extended to support other technologies.

In the future we plan to continue the development of `StreamGen` by adding new code generators for other popular streaming platforms and by providing support for currently

[17] https://beam.apache.org/

missing streaming features, such as triggers and methods for handling refinements of results. Moreover, we plan to evaluate `StreamGen` in industry to further confirm its effectiveness.

ACKNOWLEDGMENTS

This work is supported by the European Commission grant no. 644869 (H2020), DICE, by the ERC Grant agreement no. 291389, "Concurrency Made Easy" and by the European Commission gran no. 779656 (H2020), SODALITE.

We are very grateful to our anonymous reviewers for their comments and suggestions. In particular, the hints of one of these reviewers have determined a significant improvement in the definition of our UML profile.

REFERENCES

[1] Lorenzo Affetti, Riccardo Tommasini, Alessandro Margara, Gianpaolo Cugola, and Emanuele Della Valle. 2017. Defining the execution semantics of stream processing engines. *Journal of Big Data* 4, 1, 12. https://doi.org/10.1186/s40537-017-0072-9

[2] Tyler Akidau 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* (2015).

[3] João Paulo Andrade Almeida. 2006. *Model-driven design of distributed applications*. Ph.D. Dissertation. University of Twente, Netherlands. CTIT Ph.D.-Thesis Series, ISSN 1381-3617, No. 06-85 ; Telematica Instituut Fundamental Research Series, ISSN 1388-1795, No. 018.

[4] C. A. Ardagna, V. Bellandi, P. Ceravolo, E. Damiani, M. Bezzi, and C. Hebert. 2017. A Model-Driven Methodology for Big Data Analytics-as-a-Service. In *2017 IEEE International Congress on Big Data (BigData Congress)*. 105–112. https://doi.org/10.1109/BigDataCongress.2017.23

[5] Danilo Ardagna, Elisabetta Di Nitto, Giuliano Casale, Dana Petcu, Parastoo Mohagheghi, Sébastien Mosser, Peter Matthews, Anke Gericke, Cyril Ballagny, Francesco D'Andria, Cosmin-Septimiu Nechifor, and Craig Sheridan. 2012. MODAClouds: A Model-driven Approach for the Design and Execution of Applications on Multiple Clouds. In *Proceedings of the 4th International Workshop on Modeling in Software Engineering (MiSE '12)*. IEEE Press, Piscataway, NJ, USA, 50–56. http://dl.acm.org/citation.cfm?id=2664431.2664439

[6] Krishnakumar Balasubramanian, Aniruddha Gokhale, Gabor Karsai, Janos Sztipanovits, and Sandeep Neema. 2006. Developing Applications Using Model-Driven Design Environments. *Computer* 39, 2 (Feb. 2006), 33–40. https://doi.org/10.1109/MC.2006.54

[7] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. 2012. *Model-Driven Software Engineering in Practice* (1st ed.). Morgan & Claypool Publishers.

[8] G. Casale, D. Ardagna, M. Artac, F. Barbier, E. D. Nitto, A. Henry, G. Iuhasz, C. Joubert, J. Merseguer, V. I. Munteanu, J. F. PÃĆÂĪrez, D. Petcu, M. Rossi, C. Sheridan, I. Spais, and D. Vladuic. 2015. DICE: Quality-Driven Development of Data-Intensive Cloud Applications. In *2015 IEEE/ACM 7th International Workshop on Modeling in Software Engineering*. 78–83. https://doi.org/10.1109/MiSE.2015.21

[9] Mathieu Colas, Ingo Finck, Jerome Buvat, Roopa Nambiar, and Rishi Raj Singh. 2015. *Cracking the Data Conundrum: How Successful Companies Make Big Data Operational*. Technical Report. Capgemini consulting. URL: https://www.capgemini-consulting.com/cracking-the-data-conundrum.

[10] Marcos Aurélio Almeida da Silva, Andrey Sadovykh, Alessandra Bagnato, Alexey Cheptsov, and Ludwig Adam. 2014. JUNIPER: Towards Modeling Approach Enabling Efficient Platform for Heterogeneous Big Data Analysis. In *Proceedings of the 10th Central and Eastern European Software Engineering Conference in Russia (CEE-SECR '14)*. ACM, New York, NY, USA, Article 12, 7 pages. https://doi.org/10.1145/2687233.2687252

[11] Ernesto Damiani, Claudio Ardagna, Paolo Ceravolo, and Nello Scarabottolo. 2017. Toward Model-Based Big Data-as-a-Service: The TOREADOR Approach. In *Advances in Databases and Information Systems*, Mārīte Kirikova, Kjetil Nørvåg, and George A. Papadopoulos (Eds.). Springer International Publishing, Cham, 3–9.

[12] Robert Feldt and Ana Magazinius. 2010. Validity Threats in Empirical Software Engineering Research - An Initial Survey. In *Proceedings of the Software Engineering and Knowledge Engineering Conference*.

Redwood City, San Fransisco Bay, CA, USA, 374–379. http://www.cse.chalmers.se/~feldt/publications/feldt_2010_valthreats_survey.html

[13] Abel Gómez, José Merseguer, Elisabetta Di Nitto, and Damian A. Tamburri. 2016. Towards a UML Profile for Data Intensive Applications. In *Proceedings of the 2Nd International Workshop on Quality-Aware DevOps (QUDOS 2016)*. ACM, New York, NY, USA, 18–23. https://doi.org/10.1145/2945408.2945412

[14] Michele Guerriero, Alessandro Nesta, and Elisabetta Di Nitto. 2018. Streamgen: a UML-based tool for developing streaming applications. In *Proceedings of the 10th International Workshop on Modelling in Software Engineering, MiSE@ICSE 2018, Gothenburg, Sweden, May 27-28, 2018*. 57–58. https://doi.org/10.1145/3193954.3193964

[15] Ryoya Kaneko, Kohei Miyaguchi, and Kenji Yamanishi. 2017. Detecting changes in streaming data with information-theoretic windowing. In *2017 IEEE International Conference on Big Data, BigData 2017, Boston, MA, USA, December 11-14, 2017*, Jian-Yun Nie, Zoran Obradovic, Toyotaro Suzumura, Rumi Ghosh, Raghunath Nambiar, Chonggang Wang, Hui Zang, Ricardo Baeza-Yates, Xiaohua Hu, Jeremy Kepner, Alfredo Cuzzocrea, Jian Tang, and Masashi Toyoda (Eds.). IEEE Computer Society, 646–655. https://doi.org/10.1109/BigData.2017.8257980

[16] Stuart Kent. 2002. Model Driven Engineering. In *Integrated Formal Methods*, Michael Butler, Luigia Petre, and Kaisa Sere (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 286–298.

[17] Katsiaryna Labunets, Fabio Massacci, Federica Paci, Sabrina Marczak, and Flávio Moreira de Oliveira. 2018. Model Comprehension for Security Risk Assessment: An Empirical Comparison of Tabular vs. Graphical Representations. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 395–395. https://doi.org/10.1145/3180155.3182511

[18] François Lagarde, Huáscar Espinoza, François Terrier, and Sébastien Gérard. 2007. Improving UML profile design practices by leveraging conceptual domain models. In *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), Atlanta (USA)*. ACM, 445–448.

[19] Xiufeng Liu, Nadeem Iftikhar, and Xike Xie. 2014. Survey of Real-time Processing Systems for Big Data. In *Proceedings of the 18th International Database Engineering & Applications Symposium (IDEAS '14)*. ACM, New York, NY, USA, 356–361. https://doi.org/10.1145/2628194.2628251

[20] Object Management Group. 2017. OMG Unified Modeling Language – Version 2.5.1. https://www.omg.org/spec/UML/2.5.1

[21] Francesco Marconi, Marcello M. Bersani, and Matteo Rossi. 2017. A Model-driven Approach for the Formal Verification of Storm-based Streaming Applications. *SIGAPP Appl. Comput. Rev.* 17, 3 (Nov. 2017), 6–15. https://doi.org/10.1145/3161534.3161535

[22] Diego Perez-Palacin, JosÃĺ Merseguer, Jose Requeno, Michele Guerriero, Elisabetta Di Nitto, and Damian Tamburri. 2019. A UML Profile for the Design, Quality Assessment and Deployment of Data-intensive Applications. *Software & Systems Modeling* (04 2019). https://doi.org/10.1007/s10270-019-00730-3

[23] A. Rajbhoj, V. Kulkarni, and N. Bellarykar. 2014. Early Experience with Model-Driven Development of MapReduce Based Big Data Application. In *Software Engineering Conference (APSEC), 2014 21st Asia-Pacific*, Vol. 1. 94–97. https://doi.org/10.1109/APSEC.2014.23

[24] Jose Requeno, IÃśigo GascÃşn, and JosÃĺ Merseguer. 2018. Towards the Performance Analysis of Apache Tez Applications. 147–152. https://doi.org/10.1145/3185768.3186284

[25] J. Requeno, J. Merseguer, and S. Bernardi. 2017. Performance Analysis of Apache Storm Applications Using Stochastic Petri Nets. In *2017 IEEE International Conference on Information Reuse and Integration (IRI)*. 411–418. https://doi.org/10.1109/IRI.2017.64

[26] Per Runeson and Martin Höst. 2009. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering* 14, 2 (2009), 131–164.

[27] S. Santurkar, A. Arora, and K. Chandrasekaran. 2014. Stormgen - A Domain specific Language to create ad-hoc Storm Topologies. In *Federated Conference on Computer Science and Information Systems (FedCSIS)*. 1621–1628. https://doi.org/10.15439/2014F278

[28] Bran Selic. 2007. A Systematic Approach to Domain-Specific Language Design Using UML. In *Tenth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2007), 7-9 May 2007, Santorini Island, Greece*. IEEE Computer Society, 2–9.

[29] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 423–438. https://doi.org/10.1145/2517349.2522737

A IMPLEMENTING STREAMUML USING ACTIVITY DIAGRAMS: A FEASIBILITY STUDY

As previously mentioned, Activity Diagrams can be used in UML to represent the flow of data among Actions in Activities. This is accomplished by means of ObjectFlows, i.e., edges representing the transfer of objects from a source Action to a target one.

We experimented with Activity Diagrams and drafted an alternative implementation of StreamUML, called StreamUML$_{AD}$, which we then used to model the StreamingWordCount application as shown in Figure 7. Note that in this case we refer to *Transformation* and not to *Transformer*, in order to align the terminology with the semantics of Activities. In the figure, the transformations required by the example are represented as Actions properly stereotyped using the same terminology presented in Section 4.3. The data source is represented as a UML *AcceptEventAction*, stereotyped as «SocketSource». AcceptEventAction is defined directly in UML and represents a specific type of Action that waits for the occurrence of some events. An alternative way to model data sources would be to define *streaming Parameters* for an Activity and then to instantiate stereotyped *ActivityParameterNodes* (one for each data source). ActivityParameterNodes allow to model the fact that value updates for a given Parameter (which represents, in our context, the input data) may be received during the execution of the corresponding Activity. The data sink is represented as a UML *DataStoreNode*, a specific type of ObjectNode that holds objects persistently. The streams are represented by ObjectFlows connected to the source and target Actions through so called *Pins* (a specific type of ObjectNode). These last elements provide a *type* property that dictates the type of data that can flow through the stream. According to the UML specification, the type property of the downstream Pin should be of the same or a super-type of the one of the upstream Pin (in our case, we assume that these two types must be the same). The definition of two example-specific DataTypes, WordToken and WordCount, and, in general, any other complex data type the user would need to define, require the creation of the corresponding classes or data types in a Class Diagram (we do not show it here for the sake of space).

Also, in the case of Activity Diagrams, the parallelization of operators as well as the windowing and partitioning semantics associated to streams have been defined by means of new sub-stereotypes and properties as we could not exploit UML-specific features for these purposes.

For what concerns parallelism, UML does provides the notion of *ExpansionRegion*, which can be seen as a portion of an Activity including some Actions and edges (including ObjectFlows). An ExpansionRegion takes some inputs from a collection and executes its Actions for each of these inputs. The execution of Actions can happen in three different *modes*: *parallel*, which means that the execution proceeds in parallel for each input element, *iterative*, which means that when an execution in the ExpansionRegion is completed, a new one can start, and *stream*, which inhibits the parallelism [20]. Unfortunately, none of the above modes is suitable for our case as in streaming systems there is the possibility to directly control the number of parallel instances of a certain component in a way that is independent from the number of inputs. For instance, we could be willing to set three or ten parallel instances of a certain component working on 100 incoming tuples, which are then distributed among such components according to some partitioning criterion, while the parallel mode applied to an ExpansionRegion would allow us only to have 100 components working on 100 incoming tuples. For this reason, in StreamUML$_{AD}$ we could not rely on the

parallel mode defined in UML and we had to introduce explicitly a new parallelism property in our transformation operators. A similar choice has been taken in [21, 22, 24, 25].

A possible candidate to model windowed streams is the CentralBufferNode UML element, which is a specific type of ObjectNode acting as a buffer between ObjectFlows and transferring incoming object tokens (tuples in our terminology) to the outgoing flows, according to some general ordering rule. However, a CentralBufferNode, per se, does not allow any grouping of buffered elements (or object tokens) and acts as a simple mechanism to decouple the flow of objects between Actions. A way to group elements is offered by the possibility to define a *weight* for an ObjectFlow. This defines the number of tokens that "must traverse the edge at the same time" [20]. So, for instance, we could say that tuples buffered in a CentralBufferNode could be passed to a transformation as soon as they reach a certain number. This allows us to model a non-overlapping CountWindow (i.e., in the terms of the metamodel of Section 4.2, a window with *size* equal to its *slideSize*), but is not enough to cover the other windowing mechanisms. Moreover, we could not identify any way to model the data partitioning semantics, that is, the possibility to explicitly specify a criterion to be used for distributing input tokens among a set of parallel instances of the same transformation operator. For these reasons, we eventually had to introduce two sub-stereotypes of «DataStream», specifically «WindowedDataStream» and «PartitionedStream», also in StreamUML$_{AD}$.

As a results of this analysis, we have concluded that Activity Diagrams, in association with Class Diagrams for the definition of data types, are certainly a valid alternative for the purpose of creating StreamUML. However, they do not offer significant advantages compared to the combination of Class Diagrams and Composite Structure Diagrams.

First, in terms of profile elements, StreamUML$_{AD}$ is not significantly different from the other. Almost all profile elements defined in StreamUML had to be redefined in StreamUML$_{AD}$, of course, extending different meta-classes. The only exception is the possibility to reuse, in StreamUML$_{AD}$, AcceptEventActions or Parameters as data sources and DataStoreNode as a sink. However, the most complex aspects related to windowing, partitioning, and parallelism must be defined from scratch in both cases. Moreover, Composite Structure Diagrams offers a more effective multiplicity feature to express the intrinsic characteristic of parallelism in streaming applications.

Second, being Activity Diagrams richer and more expressive than InformationFlows, they offer several modeling capabilities that are not be used to model streaming applications with StreamUML$_{AD}$. This implies that `StreamGen` users should be more knowledgeable about UML than in the case the simpler alternative based on InformationFlows is adopted. For example, to model a data stream between two operators the user would need two Actions, two pins, and an ObjectFlow between the two; the *type* property of the output Pin of the source action should match that of the input Pin of the destination action, i.e., both the Pins would need to be configured in the same way. Furthermore, to model complex data types the user would need to define some DataTypes instantiating a second diagram, for instance a Class Diagram. Thus, the user would be required to know the difference between Activity Diagrams and Class Diagrams and to manage two different diagrams.

All in all, we considered that exploiting Activity Diagrams would not simplify StreamUML and, at the same time, would be more tedious for streaming application designers. Since the primary contribution of our work is `StreamGen` as a tool and a solution to simplify the development of distributed streaming applications, we decided to focus, as key requirement, on providing a simple modeling approach and an easy to use solution; this drove us towards preferring InformationFlows as the starting point for StreamUML. We acknowledge that Activity Diagrams enable the specification of low level details on how an application executes.

At the same, time the user would be required to master those details, which is in general not an easy task.

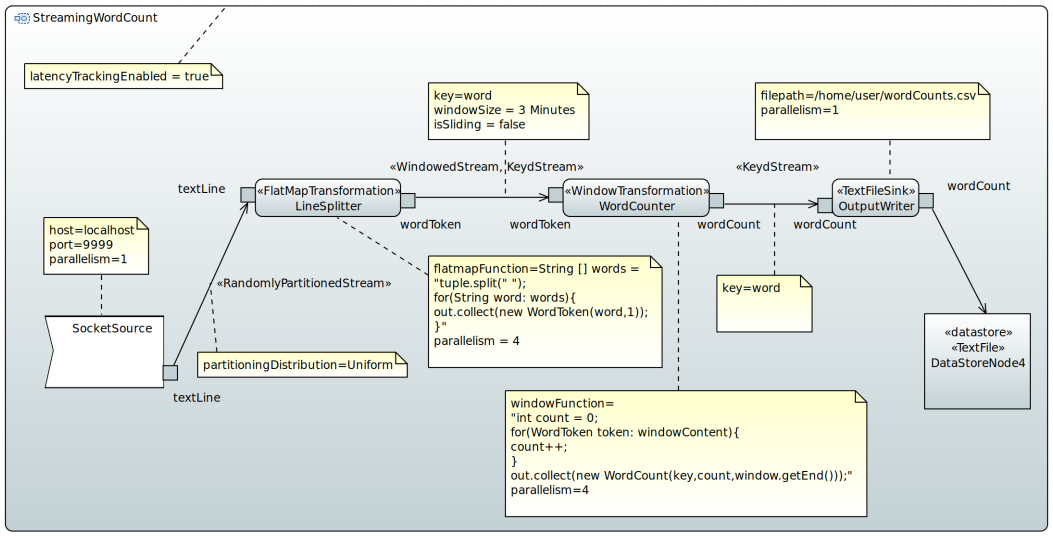


Fig. 7. Modeling the StreamingWordCount application using StreamUML$_{AD}$.