Graphical Spark Programming in IoT Mashup Tools

Tanmaya Mahapatra*, Ilias Gerostathopoulos, Christian Prehofer and Shilpa Ghanashyam Gore
* Lehrstuhl für Software und Systems Engineering, Fakultät für Informatik
Technische Universität München, Boltzmann Straße 03, 85748 Garching, Germany
Phone: +49-89-289-17840, Fax: +49-89-289-17307
Email: *mahapatr@in.tum.de,gerostat@in.tum.de,prehofer@in.tum.de, ga89top@mytum.de

***Abstract*—With the unprecedented rise in the number of IoT devices, the amount of data generated from sensors is huge and often demands an in-depth analysis to acquire suitable insights. Mashup tools, used primarily for intuitive graphical programming of IoT applications, can help both for efficiently prototyping and also data analytics pipelines. In this study, we focus on the tight integration of data analytics capabilities of Spark in IoT mashup tools. The main challenge in this direction is the presence of a wide range of data interfaces and APIs in the Spark ecosystem. In this study, we contribute to current applications by (i) providing a thorough analysis of the Spark ecosystem and selecting suitable data interfaces for use in a graphical flow-based programming paradigm, (ii) devising a novel, generic approach for programming Spark from graphical flows that comprises early-stage validation and code generation of Java Spark programs. The approach is implemented in aFlux, our JVM-based mashup tool and is evaluated in three use cases showcasing the machine learning and stream analytics capabilities of Spark.**

***Index Terms*—Internet of Things, IoT applications, data analytics, graphical flows, end-users, Spark analytics, mashup tools**

I. Introduction

In recent years, the vision of ubiquitous connected physical objects commonly referred to as the Internet of Things (IoT), has become a reality. Analytics of the IoT data generated in real-time is gaining prominence, as this leads to immediate uncovering of potentially useful insights. Big Data technologies can be employed in this context to generate insights such as end-user behavioural patterns, and applications, such as generating mobility-models for a city, predicting impending natural calamities, and performing structural health monitoring. However, the development of IoT applications is not a straightforward process because developers have to write boilerplate code to access the data sets from the sensors of different devices and also perform data mediation before actually using the data in applications. Special graphical tools called IoT mashup tools have been proposed as a way to simplify this. Mashup tools typically support a graphical interface to specify the control-flow between different sensors, services and actuators [1], [2]. The resulting application follows the flow-based programming model, where outputs from a node in the flow become inputs of the next node. An exemplary tool is Node-RED [3]. Currently, mashup tools do not support the specification and execution of flows that include Big Data analytics computations, i.e. they do not support in-flow data analytics with Big Data technologies. The underlying reasons for this limitation are (i) the single-threaded execution model, (ii) the synchronous and blocking execution semantics, and (iii) the non-concurrent execution of components in state-of-the-art IoT mashup tools (e.g. Node-RED) [4], [5].

Conversely, we have a large number of tools that are grouped together as Big Data Analytics tools. These tools are typically used for applications like targeted advertising and social network analysis. [1]. Spark is one such prominent tool, allowing for advanced, scalable, fault-tolerant analytics and comes equipped with machine learning libraries as well as stream processing capabilities [6], [7]. Nevertheless, the learning curve associated with Spark is quite steep [6]. In response to this, our study focuses on supporting Spark programming from graphical mashup tools.

To illustrate the importance of efficiently prototyping data analytics pipelines, consider a scenario where the administrator of a taxi service wishes to know the current high demand areas in the city, to redirect the available fleet accordingly and to reduce customer waiting time. Assuming that the live city data is available via REST APIs, it is still a non-trivial task to analyse and draw insights from the data since this involves data collection, data cleaning and real-time analytics to arrive at some usable conclusions. If mashup tools supported graphical blocks for each of the tasks that the administrator needs to perform, the task would become simpler as it would only involve specifying the control-flow between various graphical components. The end result is an application which does real-time Big Data analytics on sensed traffic data and uses the analytics results to re-route the taxi fleet on the fly.

The overall problem we address here is the lack of an integrated tool for IoT application development involving Big Data analytics. *aFlux*, a new actor-model based [8] mashup tool, has been developed to overcome the existing limitations of mashup tools [9]. In this paper, we present with programming Spark via graphical flows from aFlux, i.e. how to enable Spark programming at a higher level with modular components in a mashup tool. Thus, the problems can be broadly classified into two categories:

1) Diverse data-representational styles, APIs and libraries centred around Spark make it difficult to extract a common methodology with which to access the functionalities of Spark and formulate an approach to use it from a flow-based programming paradigm.
2) Reconciling the difference in the programming paradigm of Spark and flow-based mashup tools can be a chal-

lenge. Spark relies on a lazy evaluation execution model, where computations are materialised if their output is necessary, while flow-based programming has a component triggered, then proceeds to execution, and finally passes their output to the next component upon completion. To program Spark from mashup tools, this difference in the computation model needs to be addressed. Additionally, Spark is highly concurrent; however, mashups are single thread.

Succinctly, we make the following contributions corresponding to the problems enumerated above: (i) We analyse the diverse ecosystem of Spark and its various data interfaces (i.e. data abstractions) to extract a suitable data interface that would support programming Spark from the higher abstraction level of flow-based programming (Section III); (ii) We enable the modelling of Spark applications in aFlux, our JVM based mashup tool (Section IV); (iii) We evaluate the graphical programming approach in three traffic case studies involving fleet management of taxis during rush hour and discuss the limitations of our approach (Section V).

II. Background: Mashups & Big Data Analytics

A mashup application is a composite application developed through the agglomeration of reusable components. The individual components are known as 'mashup components' and they form the building blocks of the mashup application. The specification of control-flow between these mashup components forms the mashup logic. As control flows from one component to the next it typically involves potential data mediation before the data received from the preceding component can be used, as well as the execution of business logic, defined within the component. This process, performed sequentially from the first to the last component of the flow, defines the business logic of the application. Figure 1 shows a typical mashup application (created in Node-RED [3]). In addition, this also highlights the typical outlook of a mashup application, i.e. the flow-based programming paradigm it follows. The example of the figure first fetches data from a REST API, then, checks for certain conditions in the second component, and, finally, the control moves to the third component to initiate actions corresponding to the input received from its preceding component.

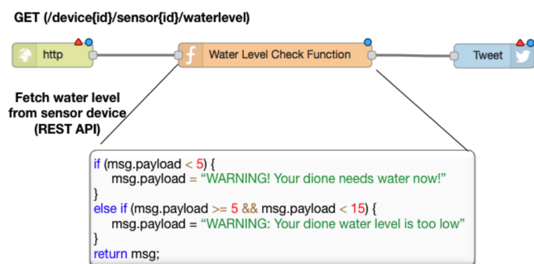

Fig. 1. Model of a mashup application in Node-RED, as in [2]

The tools which support the creation of such mashup applications are called 'mashup tools'. They typically provide a graphical interface and graphical components which can be orchestrated in a specific order. Prominent mashup tools used in IoT domain include Node-RED [3], glue.things [10], Thingstore [11], OpenIoT [12], ThingWorx [13], Paraimpu [14], and Xively [13]. It is worth noting that many such tools, providing control-specification between various visual elements, do not even refer to themselves as mashup tools. The most prominent is Node-RED, an open-source mashup tool built on Node.js by IBM. In Node-RED, a mashup application is called a 'flow" while the mashup components are called 'nodes'. Nodes are simply snippets of JavaScript code, abstracted from the end-user of the tool, that are connected diagrammatically. From a functionality perspective, these nodes can be web services, interfaces to IoT devices or abstractions for IoT protocols (e.g. MQTT reader).

Today, there are several stream-analytics engines capable of crunching and extracting business values from Big Data, including Apache Spark [6], [7], Apache Flink [15], and Kafka Streams. Spark is a scalable, fault-tolerant cluster computing framework based on the traditional MapReduce computation model [16]. Spark supports in-memory computations, thus, allowing interactive data exploration and iterative computations, like multi-pass machine learning algorithms. In addition to this, Spark supports both batch as well as stream analytics. It has evolved from a framework to an ecosystem, with several libraries built around the core framework; Spark SQL provides a SQL-like interface for data analysis, GraphX can be used for graph computations, and different Machine Learning libraries to learn from data-sets.

III. Spark Ecosystem: An Analysis

Spark supports two kinds of transformations among different libraries in its ecosystem. First, it supports high-level operators which apply user-defined methods to data, e.g. the *map* operator. The user-defined operation has to be provided by the programmer. Newer libraries of Spark have moved away from this paradigm and instead offer fine-grained operations where the operation logic is pre-fixed yet parametrizable by the programmer. Spark has introduced several abstractions to manage and organize user data within its run-time environment and several libraries have introduced data abstractions customized for different cases. Table I summarizes the libraries and the data abstractions each library requires for interaction. Fine-grained operations are possible through the Declarative APIs of Spark, i.e. all APIs of libraries listed in Table I that are based on the Spark Core library.

The Resilient Distributed Dataset (RDD) is the key abstraction for in-memory processing and fault-tolerance of the engine, which is used heavily for batch processing [17]. Higher level constructs are supplied with User-Defined Functions (UDFs) applied to the data in a parallel fashion. UDFs have to adhere to strong type checking requirements. Spark Streaming library employs the *DStream* abstraction [18], which collects data streamed over a user-defined interval and combines them

Data Interface / Library	RDD	DStream	DataFrame	S. DataFrame
Spark Core	Yes	-	-	-
Spark Streaming	-	Yes	-	-
Spark SQL	-	-	Yes	-
Spark MLib	Yes	-	-	-
Spark ML	-	-	Yes	-
Spark Structured Streaming	-	-	-	Yes

TABLE I
SPARK LIBRARIES AND DATA INTERFACES

Target Interface / Source Interface	RDD	DStream	DataFrame	S. DataFrame
RDD	-	No	Yes	No
DStream	Yes	-	No	No
DataFrame	Yes	No	-	No
Streaming DataFrame	No	No	No	-

TABLE II
INTEROPERABILITY BETWEEN DIFFERENT SPARK DATA INTERFACES

with the rest of the data received so far to create a micro-batch. This approach hides the process of combining data. *Spark Streaming* operations can be performed on a DStream abstraction or on an RDD abstraction, as DStream can be operated on by converting it to RDD. While this library is not a true stream processing library (it internally uses micro-batches to represent a stream), the most important aspect is its compatibility with *Spark MLib* library which makes it possible to apply machine learning models learned offline, on Streaming data.

The *DataFrame API*, introduced by the *Spark SQL* library, is a declarative programming paradigm for batch processing built using the *DataFrame* abstraction. This abstraction treats data as a big table with named columns, similar to real-world semi-structured data (e.g. Excel file). DataFrame API provides a declarative interface, with which data and parameters required for processing the data can be supplied. The actual implementation of the operations performed on the data to produce the desired transformation is abstracted from the user. *Spark ML* is accessed using the DataFrame API.

Finally, the *Spark Structured Streaming* library provides real-time stream processing using *Streaming DataFrame* APIs, an extension of DataFrame APIs; however. It is incompatible with the Spark ML library. This is because the incremental processing programming model of Spark Structured Streaming programming is not compatible with the Spark ML processing model, where repeated iterations are carried out on entire datasets.

Spark libraries have been built on different abstractions. The Core abstraction is RDD; the other libraries have added layers of abstraction on top of this core abstraction. Interoperability between libraries is supported in several cases as illustrated in Table II.

From the available data interfaces of Spark, DStream and DataFrame (including Streaming DataFrame), APIs are easy to represent in a flow-based programming model of mashup tools. APIs based on these prevent the usage of user-defined functions and provide predictable input and output types for each operation—the tool can then focus on validating the associated schema changes. Moreover, it is easy to represent DStream and DataFrame APIs as graphical components that can be wired together. Finally, the different input parameters required by an API can be specified by the user from the front-end.

IV. CONCEPTUAL APPROACH

The conceptual approach for programming Spark via graphical flows addresses the overall problems stated in Section I. We present the approach in two parts, corresponding to the first two contributions enumerated in Section I.

(i) Suitable data interface. For expressing the Spark semantics in a flow-based programming paradigm we proceed by selecting the data interfaces of Spark which are most suitable for wiring together. Hence, first, we define the scope of the graphical components. Users would typically drag different graphical components and wire them together in the form of a flow. In our approach, we decided to restrict ourselves to the declarative APIs of Spark as these APIs typically require some input and produce predictable outputs, making them an ideal choice as wiring components. From the tool's perspective, the input is a compatible schema and the output is a corresponding altered schema.

In contrast to this, supporting data-transformations based on user-defined functions (UDFs) would impose several challenges that are difficult to solve in a generic manner. As every UDF has tight type requirements, this would introduce type validation problems from a tool's perspective. Another consideration is these graphical components do not represent a single Spark API but rather a set of APIs, which when combined form a specific data-analytic operation. Since representing every Spark API via a graphical component would make the approach unscalable and would not provide an abstract model, our approach focusing on Declarative APIs attempts to optimally balance a proposed tool's usefulness and usability.

Additionally, since the semantics, as well as execution model of a Spark application, is different from those of mashup tools, which follow the flow-based programming paradigm, we have introduced auxiliary graphical components to express the semantics of Spark from mashup tools. These are typically used for enforcing a strict sequence of operations, e.g. when defining the order in pipeline operations of machine learning APIs or for bridging data-sets, like in join or merge operations.

(ii) Modelling of Spark flows in aFlux. Typically, a Spark application consists of three main parts, i.e. reading data from file systems or streaming sources, like IoT sensors, applying analytical transformations on those data-sets and finally writing the results to either file systems or publishing them as real-time streams, as the case may be.

One of the primary assumptions is end-users would typically follow the above high-level model while specifying a Spark application; hence, this idea can be taken as a preliminary semantic validation requirement to guide the user in creating a semantically valid application. To produce a semantically valid Spark application from the flow, the positional hierarchy of graphical components needs to be preserved. The flow is captured and represented as a directed acyclic graph (DAG), where the roots represent data read operations, branches are pathways for data transformations and leaves represent data write operations. Any node in a branch must be compatible with the schema produced by its immediate predecessor. The DAG can have multiple root nodes, since users can read data-sets from two different IoT data sources, merge them and, then, run analytics on them. In short, the DAG stores the type of graphical components used by the user and also their positional information, both of which are necessary to generate a Spark application.

We maintain method implementations of Spark operations that take a data interface schema as well as user parameters as input, make use of one or more Spark APIs to do the data transformation, and return a modified data interface schema as the output. Every node in the DAG typically corresponds to one Spark operation and, thus, to one standalone method implementation. Hence, the captured DAG is passed to a code generator, which first generates the necessary code skeleton for initializing a Spark session and then closes the session at the end of the application, to create the runnable Spark application. For the actual business logic of the flow, it wires the method implementations of Spark operations by providing the data interface schema and user parameters as inputs. The only requirements for this wiring process are that the data interface provided as an input is the same as the data interface of the output of the previous method, and the data interface schema must be compatible.

From a high-level perspective, the process of programming Spark graphically consists of validating the flow to ensure it follows the semantics of Spark and translating the flow into a Spark application. Figure 2 illustrates the key concepts for creating a Spark job involving machine learning algorithms or real-time analytics, by creating a unidirectional flow of connected components/graphical blocks.

To support graphical Spark flows, we have used the high-level design decisions discussed above to create components for aFlux, our JVM based mashup tool. They consist of five basic component types namely *input*, *transformation*, *bridge*, *action* and *executor*. Components of these types form the nodes in the DAG. Input components have no incoming connections and they read data from external sources into the run-time environment. From the user's perspective these start a Big-Data processing flow, i.e. they are the first components in a flow and their output is consumed by other components. From the mashup tool's perspective, they introduce the schema to be used by succeeding components in the flow. Transformation components are intermediate components and represent operations on ingested data; they consume as well

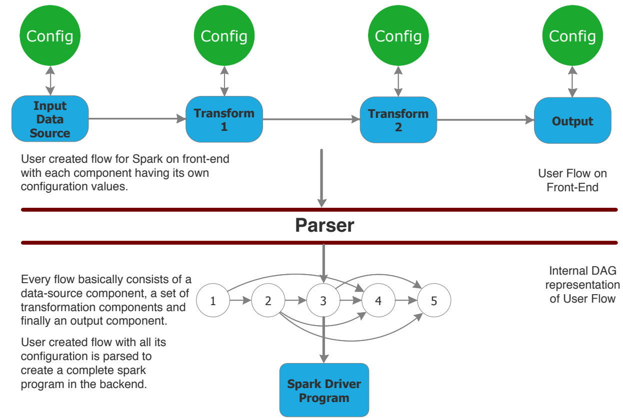

Fig. 2. High-level view of key concepts: designing of Spark jobs via graphical flows

as produce schema. Transformation APIs in Spark is designed to accept, at most, two compatible schema variants, which means they can accept, at the most, two incoming connections; they must also have at least one outgoing connection. Bridge components do not consume or produce schema. Accordingly, they do not correspond to any method implementation of Spark operation but are used to express the Spark semantics in a flow-based programming paradigm. For example, they can impose an order in processing data coming from preceding transformations. Action components allow the user to save the transformed data to external file systems or to stream it out to message distribution systems. Finally, an executor component collects all the incoming connections from multiple action components and adds abstractions related to Spark Driver program. aFlux has all the data required to generate a Spark application only after the executor component has been triggered.

Based on the classification of components, a Spark flow needs to adhere to the following rules:

1) Flow is unidirectional. Every branch in a flow begins with an input component, followed by one or more transformation components which must lead to one and only one action component.
2) Every flow must end with only one executor component and each action component in the flow must be connected to the executor component.
3) Transformation components, which require incoming connections in specific order, must be preceded by bridge component(s).
4) A component accepts an incoming connection(s) if and only if schema derived from the incoming connection(s) is valid against schema checks. This means that a named column operated upon in a component is part of its incoming schema.
5) Each component internally uses one Spark data interface which is represented by a uniform colour code. A flow

composed of different coloured components, with the exception of the executor component, is not accepted.

Every Spark component in aFlux has (i) colour denoting the data interface the component uses internally, (ii) a unique name used to represent it in the internal model, (iii) category, i.e. either input, transformation, action, execute or bridge, (iv) a set of user-supplied parameters, (v) internal logic, which houses the core functionality of the component. Hence, every component interacts with the end-user designing the flow and also contributes to creating an internal model of the user flow. All the components in a Spark flow are validated based on their position in the flow, i.e. if they are allowed in a specific position in the flow, compatibility of the data interfaces and if the input schema is compatible with them. For validation, we maintain meta-data storing the category of every aFlux component and its list of permissible predecessors. If the validation is passed, then, an internal model of the user flow is created from which the generation of a runnable Spark application proceeds.

Every Spark aFlux component has a unique internal name and an associated standalone method implementation for an analytic operation. Meta-data of this information are maintained on the tool level. The code-generator receives the DAG as input and generates the required static code, e.g. starting a Spark session, inclusion of Java packages. Then, it checks the node in the DAG and determines its category. If it is an input or transformation node, then, it uses the node's internal name to determine its associated method implementation. It calls the method via Java Reflection, passing the data interface schema and the node's user-supplied configuration values as parameters. This process is done iteratively for all nodes in the DAG until the code generator reaches the leaf nodes, which indicate actions and terminate the flow. Here, it simply calls the appropriate method to publish the data and closes the Spark session. The resulting Java file is compiled into a runnable Spark application and deployed on a cluster. Based on this conceptual approach, we have implemented Spark SQL, Spark ML, Spark Structured Streaming components in aFlux based on the DataFrame interface and Spark Streaming components based on the DStream interface.

V. Evaluation

The evaluation scenario has been designed to capture the modularity of this approach, code-abstraction from end-user, automatic handling of boilerplate code as well as interconversion of data between different data interfaces of Spark and ease of creating quick Spark jobs in the context of IoT. Here we take an example of taxi fleet management with three use-cases: (i) producing a machine-learning model to learn traffic conditions, (ii) applying the model to streaming data to make decisions, (iii) performing aggregations on streaming data. In all three use-cases, the case of manually programming them in Java has been compared with the specification of graphical flows using Spark components of aFlux. Then, we describe in detail how various aFlux components abstract the necessary steps used by programmers while developing a Spark application from scratch.

The dataset in a traffic scenario consists of information that was published by a vehicle at the beginning of a new trip. The dataset contains three elements: time-stamp, latitude and longitude. The time-stamp records the time at which a new trip commenced; latitude and longitude identify the geographic coordinates from where the new trip commenced. The goal is to devise a machine-learning based rush-hour fleet management solution to reduce waiting time for customers using Spark. Machine learning is employed to partition the city into sectors using historical data. Thus, the model prepared remains on the disk, which is then applied to real-time streaming data. Finally, stream aggregations, such as window count and running count based on event time, are applied to streaming data to receive real-time updates. The idea is to demonstrate how users can develop Spark applications for three different Spark libraries via aFlux vis-a-vis programming the same solution manually. Development of a Spark application consists of identifying relevant Spark libraries and using relevant APIs to build the solution. For example, a KMeans Algorithm is a good choice for identifying trip start hotspots. In our dataset, $<$*latitude,longitude*$>$ can be used as features for training a KMeans algorithm. The model trained on historical data should be applied to real-time data. The Pipeline API from the Spark ML library is a good choice for building a re-usable model which can persist on external file systems. Since Spark ML is built on the Spark SQL engine, using DataFrame API is the natural choice for this application. Spark libraries, which handle streaming data, support applying persisting models on real-time data and support event-time based window aggregations on streaming data. Spark Structured Streaming is a good choice for performing aggregations as it supports event-time based windowed processing. However, the programming model of Spark Structured Streaming is not compatible with Spark machine learning libraries. Hence, Spark Streaming must be used to apply the created model to real-time data and Spark Structured Streaming must be used to perform aggregations. Therefore, we selected: (i) Spark ML for developing re-usable K-Means Model, (ii) Spark Streaming for applying model on real-time data and (iii) Spark Structured Streaming: for applying aggregations on real-time data.

A. *Programming in Spark*

We now present applications programmed manually in Java. The Spark application for producing a machine learning model begins with initializing a Spark session, after which input data are read using the reader function of Spark session. The DataFrame API views the data as a table. Hence, the schema must be supplied by the reader. Listing 1 shows the relevant code for this step. Then, we apply an ML algorithm to transform the data and fit a KMeans model to it. We select the features necessary for training the KMeans algorithm and use the VectorAssembler API, which creates a field of vector data-type out of one or more fields present in the DataFrame. The Pipeline API is used to prepare a data analytics sequence:

VectorAssembler, for data preparation, followed by KMeans, for data analysis. A model is created and persisted. Listing 2 shows the relevant code that a programmer must write. The transformed data is visualized using ‘show API’. Finally, the Spark session is closed. This is approximately 50 lines of Java code.

```
//read data from external file system
Dataset<Row> inputDataSet = spark.read()
.format("csv")
.option("header", "false")
.option("mode", "DROPMALFORMED")
.option("inferSchema","false")
.schema(csvschema)
.load("trip-data.csv");
```

Listing 1. Reading Data using DataFrame API

```
String[] inputCols = {"latitude","longitude"};
// Feature Extraction
PipelineStage myAssembler = new VectorAssembler()
.setInputCols(inputCols)
.setOutputCol("features");

//ML Algorithm: apply KMeans algorithm
PipelineStage kmeans = new KMeans()
.setK(8)
.setFeaturesCol("features")
.setPredictionCol("Prediction");

//Add stages into a pipeline
PipelineStage[] stage = {myAssembler,kmeans};
Pipeline myPipeline = new Pipeline().setStages(stage);

// Model fitting
PipelineModel model = myPipeline.fit(inputDataSet);
```

Listing 2. Applying KMeans Algorithm

For the second use-case, i.e. stream processing, using Spark Streaming is a bit complicated since the Spark Streaming library is built on the Spark Core and data interface provided is from DStreams. Since Spark Core, treats all data as though it were unstructured data, this involves many steps of data interface format conversions. The program begins with a Spark Streaming session, created from a Spark session which requires the duration of a micro-batch as one of the inputs. Next, streaming data is read from Kafka in the form of <*key,value*> pairs and converted into a JavaPairDstream data interface, as shown in Listing 3. For transformation, applying a model produced using the Pipeline API on the DStream data interface is not possible. Hence, we convert DStreams collected over the micro-batch duration into RDD. Each RDD is transformed into DataFrame and the created ML model from the first use-case is applied. The manual code for this step is quite exhaustive and has not been included due to page limitations. Finally, we push the results back to Kafka and terminate the Spark Streaming session. This takes approximately 75 lines of Java code.

```
//Produce DStream from Kafka Record
JavaInputDStream<ConsumerRecord<String, String>> stream =
KafkaUtils.createDirectStream(ssc, LocationStrategies.PreferConsistent(),
ConsumerStrategies.<String,String>Subscribe(topics,kafkaParams));

JavaPairDStream<String, String> input = stream.mapToPair(
record -> new Tuple2<(record.key(),record.value()));

JavaDStream<String> inputStream =input.map(new Function<Tuple2<String,String>,
        String >(){
public String call(
Tuple2<String, String> tuple2){
return tuple2._2();}
});
```

Listing 3. Conversion of Kafka $< key, value >$ pairs to JavaPairDStream

In the final use-case, we perform stream aggregations based on event-time and windowed over a given duration. Data is read from Kafka. The Spark Structured Streaming library comes with a built-in Kafka reader, which reads Kafka messages in JSON format and maps them to the supplied schema. Then, it is converted to a DataFrame format. Next, windowed stream aggregations based on event-time are applied to the streaming data, with a watermark for handling late data. The results are sent back to Kafka. This takes approximately 45 lines of Java code.

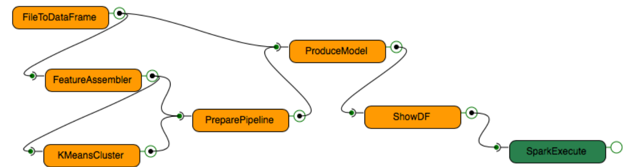

Fig. 3. Spark flow in aFlux for produce Machine Learning model

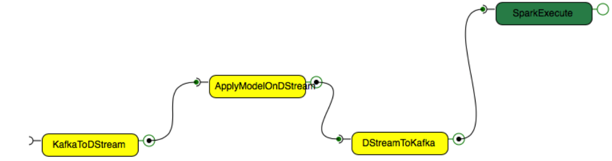

Fig. 4. Spark flow in aFlux for applying model to Streaming data

B. *Programming via aFlux*

In this section, we develop Spark applications for the same use-cases by creating graphical flows in aFlux and the Spark application is generated automatically by the mashup tool. Figure 3 shows the graphical flow for creating a machine learning model. The first aFlux component in the flow, i.e. *‘FiletoDataFrame’* has a configuration panel where the user can specify the file type, its location and what fields to read. This component abstracts away the code necessary to read the file and to convert it to a DataFrame. The *‘FeatureAssembler’* component selects the fields necessary for model training, while the *‘KMeans Cluster’* component abstracts the real K-Means algorithm. *‘PreparePipeline’* is a bridge component necessary for ensuring the execution order of *‘FeatureAssembler’* and *‘KMeans Cluster’*. Finally, the model is prepared with the *‘Produce Model’* component and the result is saved. The last aFlux component *‘Spark Execute’* marks the end of the graphical flow. Its encounter in the flow conveys a special meaning to the translator i.e. to generate the Spark session initialization, as well as the termination codes necessary for the Spark application.

Following exactly the above steps, the graphical flows for deploying a machine learning model on real-time data is shown in Figure 4. Here, the first component, *‘KafkatoDStream’* reads data from Kafka and has the necessary conversion code to transform the data into DStream. The second component, *‘Apply Model on DStream’* takes the path of a previously created Machine Learning model and applies it to the input DStream data-set and the result is moved back to Kafka by the third component in the flow, *‘DStreamtoKafka’*. This com-

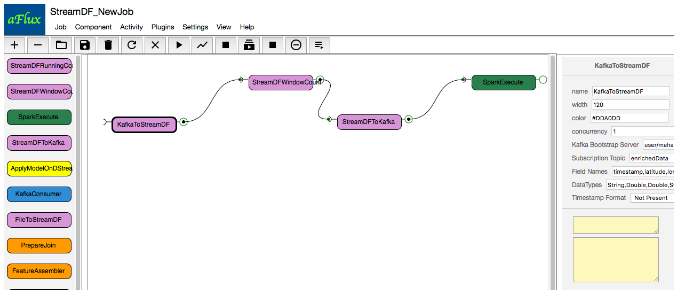

Fig. 5. Spark flow in aFlux for streaming aggregations

ponent does the automatic conversion of data from DStreams to <*key,value*> format for Kafka.

Similarly, the flow for performing streaming aggregations on real-time data is shown in Figure 5 where the first component, *'KafkatoStreamDF'* reads <*key,value*> from Kafka and converts it into streaming DataFrame automatically. The second component takes the kind of window-aggregation to be performed as user-input and applies it to the incoming data-set. The result is automatically converted to <*key,value*> format and pushed to Kafka by the *'StreamDFtoKafka'* component.

On deployment of the flows, aFlux instantiates an actor for each of the aFlux components. The actors pass messages to the next connected actors to build a DAG out of the created flow and also, each aFlux component is checked for potential semantic errors, based on its position and the type of data interface it is using internally. Every node, apart from the nodes coming in the category of bridge components in the DAG, represents a Spark operation that has a corresponding standalone method implementation of the relevant Spark APIs. The DAG is passed to a code generator, which generates a complete Spark application, and compiles and deploys it on the cluster for execution.

C. Discussion

The approach of creating Spark jobs via graphical flows, as described above, abstracts the code generation from end-users and handles data interface interconversion automatically. Automatic semantic checks ensure that the user creates a semantically valid flow that can be translated into a runnable Spark application. This enables quick prototyping of Spark applications, which is not necessarily limited to IoT use-cases. It lowers the learning curve for Spark users and abstracts away the complexities, thereby helping users to focus on the business logic. The goal of this work was to identify the most suitable data interfaces of Spark that can be modelled in a flow-based programming paradigm. Declarative APIs based on Data Frame and DStream were found to be the most suitable, as they eliminate user-defined functions to bring about data transformations. They accept compatible input schema and parameters for the transformational logic embedded within the API. aFlux components representing Spark operations have been modelled using APIs from Spark SQL, Spark ML, Spark Structured Streaming and Spark Streaming libraries. The approach used to model a Spark application is based on three distinct and successive processes, i.e. load data, transform data and finally publish the result. This concept also enforces semantic requirements of the flow, i.e. if a particular component is allowed at a particular position or not, and its compatibility with the preceding input schema. Accordingly, all aFlux components have been classified either as input, action or transformational components. Additionally, all the components in a flow must use the same data interface of Spark for a semantically valid flow. Once this is ensured, the user-flow is captured as a DAG where the nodes represent specific Big Data operations, making use of one or more Spark APIs. We maintain a standalone method implementation of the Big Data operations, which takes a data interface schema and user parameters as inputs and give a modified data interface schema as output. The code generator takes the DAG as input and generates a complete Spark application by generating necessary code, like initializing a Spark session and then wiring the standalone method implementations using Java Reflection, while passing the user parameters and the data interface schema from the preceding node in the DAG as input. The result is a runnable Spark program, compiled, packaged and deployed on a Spark cluster. This approach does not support all Spark libraries and needs interconversion of data interface format where such operations are required, e.g. applying a machine learning model produced using the Pipeline API cannot be applied on real-time data from Kafka which is typically captured as DStream. Limitations or non-supportability of Spark operations arise where the selected data interfaces, i.e. Data Frame and DStream, lack interconversion compatibility as listed in Table II.

VI. Related Work

We did not find any mashup tool which allows wiring components to produce a Spark application. However, there are many different solutions to reduce the challenges involved in using Spark. One of the closest existing solutions is Lemonade, which aims to provide data engineers with a platform to develop visual flows for producing Spark applications in Python [19]. However, its support is limited to the APIs of MLib library and does not support streaming analytics. Similarly, Apache Zeppelin provides an interactive environment for using Spark, instead of writing a complete Spark application. Zeppelin manages a Spark session within its run-time environment and interacts with Spark in interactive mode [20], while consuming code snippets in Python/R. Nevertheless, to successfully interact with Spark, Zeppelin still requires programming skills from users since it requires compilable code. A third solution is Azure, a private cloud computing service offered by Microsoft, which also offers Spark as a service. Users can configure a Spark cluster without requiring any manual installation. Here, Spark can be used to run interactive queries, visualize data and run machine learning algorithms [21]. Nevertheless, it expects the user to have programming expertise. IBM SPSS Modeller provides a graphical user interface to develop data analytics flows involv-

ing simple statistical algorithms, machine learning algorithms, data validation algorithms and visualization types [22]. SPSS Modeller provides machine learning algorithms developed using Spark MLib library which can be launched on Spark cluster by simply connecting them as components in a flow. Although SPSS Modeller is a tool built for non-programmers to perform data analytics using pre-programmed blocks of algorithms, it does not support wiring new Spark applications. Finally, Apache NiFi [23], a tool for creating data pipelines in the form of visual flows, supports integration with Spark represented by a GUI component called a Spark processor. Nevertheless, this processor needs to contain Spark code. From the perspective of an end-user, NiFi does not reduce the programming challenges associated with Spark, although automation via data pipelines is certainly provided.

VII. Conclusion

We have discussed the importance of Big Data analytics in the context of IoT and why enabling Spark programming from mashup tools is an important contribution. Accordingly, (i) we thoroughly analysed the Spark ecosystem and found the declarative APIs based on DataFrame and DStream data interfaces to be the most suitable candidates for use in a graphical flow-based programming paradigm i.e. mashup tools; (ii) we devised a novel generic approach for programming Spark from graphical flows. The conceptual approach was implemented in aFlux, our JVM based mashup tool and evaluated against three specific Spark operations i.e. creating a machine learning model, applying a machine learning model to real-time data and performing streaming aggregations on real-time data. All these usages were done manually via Java, as well as aFlux. Ease of use, code-abstraction and automatic data interface conversion, key in lowering the learning curve of Spark, were also demonstrated.

Acknowledgement

This work is part of the TUM Living Lab Connected Mobility (TUM LLCM) project and has been funded by the Bavarian Ministry of Economic Affairs, Energy and Technology (StMWi) through the Center Digitisation.Bavaria, an initiative of the Bavarian State Government.

References

[1] T. Mahapatra, I. Gerostathopoulos, and C. Prehofer, "Towards integration of big data analytics in internet of things mashup tools," in *Proceedings of the Seventh International Workshop on the Web of Things*, ser. WoT '16. New York, NY, USA: ACM, 2016, pp. 11–16. [Online]. Available: http://doi.acm.org/10.1145/3017995.3017998

[2] C. Prehofer and I. Gerostathopoulos, "Modeling restful web of things services: Concepts and tools," in *Managing the Web of Things*, 2017.

[3] "IBM Node-RED, A visual tool for wiring the Internet of things." [Online]. Available: http://nodered.org/

[4] J. Mineraud, O. Mazhelis, X. Su, and S. Tarkoma, "A gap analysis of internet-of-things platforms," *CoRR*, vol. abs/1502.01181, 2015. [Online]. Available: http://arxiv.org/abs/1502.01181

[5] "Project consortium tum living lab connected mobility: Digital mobility platforms and ecosystems," Software Engineering for Business Information Systems (sebis), München, Tech. Rep., Jul 2016. [Online]. Available: https://mediatum.ub.tum.de/node?id=1324021;

[6] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, A. Ghodsi, J. Gonzalez, S. Shenker, and I. Stoica, "Apache spark: A unified engine for big data processing," *Commun. ACM*, vol. 59, no. 11, pp. 56–65, Oct. 2016. [Online]. Available: http://doi.acm.org/10.1145/2934664

[7] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1863103.1863113

[8] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular actor formalism for artificial intelligence," in *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, ser. IJCAI'73. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1973, pp. 235–245. [Online]. Available: http://dl.acm.org/citation.cfm?id=1624775.1624804

[9] T. Mahapatra, C. Prehofer, I. Gerostathopoulos, and I. Varsamidakis, "Stream Analytics in IoT Mashup Tools," in *Proceedings of the Visual Languages and Human Centric Computing*, ser. VL/HCC, 2018, p. In Press.

[10] R. Kleinfeld, S. Steglich, L. Radziwonowicz, and C. Doukas, "glue.things: A Mashup Platform for wiring the Internet of Things with the Internet of Services," in *Proceedings of the 5th International Workshop on Web of Things*, ser. WoT '14. ACM, 2014, pp. 16–21. [Online]. Available: http://doi.acm.org/10.1145/2684432.2684436

[11] K. Akpinar, K. A. Hua, and K. Li, "Thingstore: A platform for internet-of-things application development and deployment," in *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '15. ACM, 2015, pp. 162–173. [Online]. Available: http://doi.acm.org/10.1145/2675743.2771833

[12] J. Kim and J. W. Lee, "OpenIoT: An open service framework for the internet of things," in *Internet of Things (WF-IoT)*, March 2014, pp. 89–93.

[13] H. Derhamy, J. Eliasson, J. Delsing, and P. Priller, "A survey of commercial frameworks for the internet of things," in *ETFA*, Sept 2015, pp. 1–8.

[14] A. Pintus, D. Carboni, and A. Piras, "Paraimpu: a platform for a social web of things," in *Proceedings of the 21st international conference companion on World Wide Web*. ACM, 2012, pp. 401–404. [Online]. Available: http://dl.acm.org/citation.cfm?id=2188059

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

[16] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251254.1251264

[17] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing." in *NSDI*, S. D. Gribble and D. Katabi, Eds. USENIX Association, 2012, pp. 15–28.

[18] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: ACM, 2013, pp. 423–438. [Online]. Available: http://doi.acm.org/10.1145/2517349.2522737

[19] W. dos Santos, L. F. M. Carvalho, G. de P. Avelar, Á. S. Jr., L. M. Ponce, D. O. Guedes, and W. M. Jr., "Lemonade: A scalable and efficient spark-based platform for data analytics," in *CCGrid*. IEEE Computer Society / ACM, 2017, pp. 745–748.

[20] "Apache Zeppelin," https://zeppelin.apache.org/docs/0.7.0/, [Online; accessed 22-June-2018].

[21] "Microsoft Azure," https://docs.microsoft.com/en-us/azure/hdinsight/, [Online; accessed 22-June-2018].

[22] "IBM SPSS Modeller," https://www.ibm.com/products/spss-modeler, [Online; accessed 22-June-2018].

[23] "Apache NiFi," https://nifi.apache.org/docs.html, [Online; accessed 05-September-2018].