Designing Flink Pipelines in IoT Mashup Tools*

Tanmaya Mahapatra[1], Ilias Gerostathopoulos, Federico Alonso Fernández Moreno, and Christian Prehofer

Lehrstuhl für Software und Systems Engineering, Fakultät für Informatik, Technische Universität München
tanmaya.mahapatra@tum.de

Abstract. Internet of Things (IoT) applications are generating increasingly large amounts of data because of continuous activity and periodical sensing capabilities. Processing the data generated by IoT applications is necessary to derive important insights—for example, processing data from CO emissions can help municipal authorities apply traffic restrictions in order to improve a city's air quality. State-of-the-art stream-processing platforms, such as Apache Flink, can be used to process large amounts of data streams from different IoT devices. However, it is difficult to both set-up and write applications for these platforms; this is also manifested in the increasing need for data analysts and engineers. A promising solution is to enable domain experts, who are not necessarily programmers, to develop the necessary stream pipelines by providing them with domain-specific graphical tools. We present our proposal for a state-of-the-art mashup tool, originally developed for wiring IoT applications together, to graphically design streaming data pipelines and deploy them as a Flink application. Our prototype and experimental evaluation show that our proposal is feasible and potentially impactful.

Keywords: Flink pipelines · graphical tool · IoT mashup tools · stream analytics

1 Introduction

In recent years, there has been an upsurge in the number and usage of ubiquitous connected physical devices, thereby making the era of the Internet of Things (IoT) a reality. IoT is defined as the interconnection of ubiquitous computing devices for increased value to end users [4]. Realising this value of IoT for end-users depends heavily on its software applications which in turn depends on the insights gained from IoT data.

IoT data typically comes in the form of data streams that often need to be processed under latency requirements to obtain insights in a timely fashion. Examples include traffic monitoring and control in a smart city; traffic data from different sources (e.g. cars, induction loop detectors, cameras) need to be combined in order to take traffic control decisions (e.g. setting speed limits,

* Copyright held by the author(s). NOBIDS 2018

opening extra lanes in highways). The more sensors and capabilities, the more data streams require processing.Specialised stream-processing platforms, such as Apache Flink, Spark Streaming and Kafka Streams, have been proposed to address the challenge of processing vast amounts of data (also called Big Data), that come in as streams, in a timely, cost-efficient and trustworthy manner.

The problem with existing stream platforms is that they are difficult to both set-up and write applications for. The current practice relies on human expertise and the skills of data engineers and analysts, who can deploy Big Data stream platforms in clusters, manage their life-cycle and write data analytics applications in general-purpose high-level languages such as Java, Scala and Python. Although many platforms, including Flink and Spark, provide SQL-like programming interfaces to simplify data manipulation and analysis, the barrier is still high for non-programmers.

In response to this growing need, we believe a promising solution is to enable domain experts, who are not necessarily programmers, to develop the necessary pipelines for streaming data analytics by providing them with domain-specific graphical tools. In particular, we propose to extend existing flow-based graphical programming environments, used for simplifying IoT application development, called IoT mashup tools, and allow the specification of streaming data analytics pipelines (programs) via their intuitive graphical interfaces which allow components to be dragged , dropped and wired together.

To provide a technical underpinning for our proposal and evaluate its feasibility, we have extended aFlux[1] [11, 10], a state-of-the-art mashup tool developed in our department, to support the specification of streaming data pipelines for Flink, one of the most popular Big Data stream-processing platforms. One main challenge is reconciling the difference in Flink's programming paradigm and flow-based mashup tools. Flink relies on a lazy evaluation execution model, where computations are materialised if their output is necessary, while flow-based programming triggers a component, proceeds to execution and finally passes their output to the next component upon completion. To program Flink from mashup tools, the difference in the computation model needs to be addressed. Additionally, there needs to be a seamless connection between the two systems to enable a smoother consumption of the generated results.

Succinctly, we provide the following contributions in this paper:

1. We analyse the Flink ecosystem and identify the abstractions that will work for graphical programming of Flink pipelines (Section 3).
2. We describe the concept idea and technical realisation of mapping a graphical flow, designed in aFlux, to a Flink pipeline and providing basic flow validation functionalities at the level of aFlux (Section 4).
3. We evaluate our proposal by designing pipelines that monitor traffic conditions and detect patterns in the incoming streaming data using real-time traffic data from the city of Santander, Spain (Section 5).

[1] https://github.com/mahapatra09/aflux_tum

2 Background

In this section we give an overview of mashup tools, with an emphasis on aFlux and Big Data stream analytics platforms, with emphasis on Flink.

2.1 aFlux: An IoT Mashup Tool

Mashups are a conglomeration of several accessible and reusable components on the web [5]. Mashup tools simplify the development of mashups by allowing end-users to wire together mashup components, encapsulating business logic into one or more mashup flows. When executing a mashup flow, control follows the data flow from one component to the next; this type of flow-based programming paradigm is also followed in a very popular mashup tool for IoT, Node-RED [7, 1].

aFlux is a recently proposed IoT mashup tool that offers several advantages compared to Node-RED. It features a multi-threaded execution model, asynchronous and non-blocking execution semantics and concurrent execution of components.

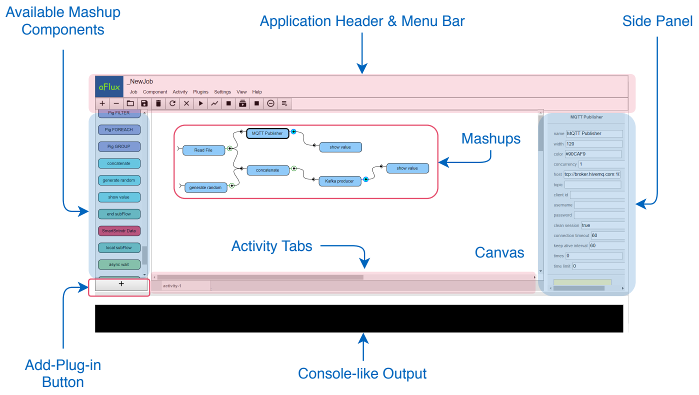


Fig. 1: Graphical User Interface of aFlux

aFlux consists of a web application and a back-end developed in Java and the Spring Framework[2]. The web application is composed of two main entities: the front-end and back-end, based on REST API. The front-end of aFlux (Fig. 1) provides a GUI for the creation of mashups. It is based on React[3] and Redux[4] frameworks. Mashups are created by dragging-and-dropping mashup components

[2] https://spring.io/
[3] https://reactjs.org/
[4] https://redux.js.org/

from the left panel. New mashup components are loaded from *plug-ins*. The application shows a console-like output in the footer, and the details about a selected item are shown on the right panel. Using the aFlux front-end, a user can create a flow by wiring several mashup components (or sub-flows) together.

When a flow is sent to the back-end, it is translated to an internal model, which is a graph called the *'Flow Execution Model'* [10]. This model is composed of *actors*, as aFlux makes use of Akka actor system[5] and Java. In an *actor system*, actors encapsulate both a state and behaviour. When an actor receives a message, it starts to perform the associated computations, and it may send a message to another actor when finished. In aFlux, a mashup component of the front-end corresponds to an actor in the back-end. Messages can only be sent asynchronously between actors [10]; concurrency of actors is also supported. Currently, aFlux supports graphical Spark programming by making use of the declarative APIs of the Spark eco-system [10].

2.2 Stream Analytics

The idea of processing data as streams, i.e. as they come in, is different from batch processing. The latter approach was followed in the first Big Data-processing systems, such as in Hadoop's MapReduce and in Apache Spark, which mainly dealt with reliable parallel processing of Big Data residing in distributed file systems, such as Hadoop's HDFS. Stream processing of Big Data has been recently sought as a solution to reduce the latency in data processing and provide real-time insights (e.g. on the scale of seconds or milliseconds).

In particular, an ideal stream-processing platform should meet the following requirements [15]:

- **Low latency**. Streaming platforms usually make use of *in-memory* processing, in order to avoid the time required to read/write data in a storage facility and thus decrease the overall data-processing latency.
- **High throughput**. Scalability and parallelism enable high performance in terms of data-processing capability. The real-time performance of stream-processing systems is frequently demanded even with spikes in incoming data [6].
- **Data querying**. Streaming platforms should make it possible to find events in the entire data stream. Typically, SQL-like language is employed [15]. However, since data streams never end, there needs to be a mechanism to define the limits of a query; otherwise it would be impossible to query streaming data. This is where the *window* concept takes part. Windows define the data in which an operation may be applied, so they become key elements in stream-processing.
- **Out-of-order data**. Since a streaming platform does not wait for all the data to become available, it must have a mechanism to handle data coming late or never arriving. A concept of *time* needs to be introduced, to process data in chunks regardless of order of arrival.

[5] https://akka.io/

- **High availability and scalability**. Stream processors will most likely handle ever-growing amounts of data, and in most cases, other systems could rely on them, e.g. in IoT scenarios. For this reason, the stream-processing platform must be reliable, fault-tolerant and capable of handling any amount of data events.

The first approaches to stream processing, notably Storm and Spark Streaming, used to focus on requirements such as low latency and high throughput [8]. *Lambda architecture*, a well-known approach [6, 12, 9] combines batch and stream-like approaches to achieve shorter response times (on the order of seconds). This approach has some advantages, but one critical downside: the business logic needs to be duplicated into the stream and the batch processors. In contrast to this, stream-first solutions, such as Apache Flink, meet all the outlined requirements [6].

3 Flink Ecosystem: An Analysis

Apache Flink is a processing platform for distributed stream as well as batch data. Its core is a streaming data-flow engine, providing data distribution, communication and fault tolerance for distributed computations over data streams [16]. It is a distributed engine, built upon a distributed runtime that can be executed in a cluster to benefit from high availability and high-performance computing resources. It is based on stateful computations. Indeed, Flink offers exactly-once state consistency, which means it can ensure correctness even in the case of failure. Flink is also scalable because the state can be distributed among several systems. It supports both bounded and unbounded data streams. Flink achieves all this by means of a distributed data-flow runtime that allows a real-stream pipelined processing of data.

A streaming platform should be able to handle time because the reference frame is used for understanding how the data stream flows, that is to say, which events come before or after another. Time is used to create windows and perform operations on streaming data, in a broad sense. Flink supports several concepts of time: (i) Event time refers to the time at which an event was produced in the producing device. (ii) Processing time is related to the system time of the cluster machine in which the streams are processed. (iii) Ingestion time is the wait time between when an event enters the Flink platform and the processing time.

Windows are a basic element in stream processors. Flink supports different types of windows, and all of them rely on the notion of time as described above. Tumbling windows have a specified size, and they assign each event to one and only one window without any overlap. Sliding windows have fixed sizes, but an overlap, called the slide, is allowed. Session windows can be of interest for some applications, because sometimes it is insightful to process events in sessions.A global window assigns all elements to one single window. This approach allows for the definition of triggers, which tell Flink exactly when the computations should be performed.

The Flink distributed data-flow programming model together with its various abstractions for developing applications, form the Flink ecosystem. Flink offers three different levels of abstraction to develop streaming/batch applications as follows: (i) Stateful stream processing: The lowest level abstraction offers stateful streaming, permitting users to process events from different streams. It features full flexibility by enabling low-level processing and control. (ii) Core level: above this level is the core API level of abstraction. By means of both a DataStream API and a DataSet API, Flink enables not only stream processing but also batch analytics on ‘bounded data streams’, i.e., data sets with fixed lengths (iii) Declarative domain-specific language: Flink offers a Table API as well, which provides high-level abstraction to data processing. With this tool, a data set or data stream can be converted to a table that follows a relational model. The Table API is more concise, because instead of the exact code of the operation, defined logical operations [16] are less expressive than the core APIs. In the latest Flink releases, an even-higher-level SQL abstraction has been created as an evolution of this declarative domain-specific language. In addition to the aforementioned user-facing APIs, some libraries with special functionality are built. The added value ranges from machine learning algorithms (currently only available in Scala) to complex event processing (CEP) and graph processing.

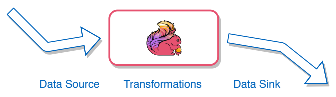


Fig. 2: Overall Structure of a Flink Program [16]

The structure of a Flink program (especially when using the core-level APIs) begins with data from a source entering Flink, where a set of transformations is applied (window operations, data filtering, data mapping, etc.). The results are subsequently yielded to a data sink, as shown in Figure 2. A Flink program typically consists of streams and transformations. Simplistically, a stream is a never-ending flow of data-sets, and a transformation is an operation on one or more streams that produces one or more streams as output.

On deployment, a Flink program is mapped internally as a data-flow consisting of streams and transformation operators. The data-flow typically resembles directed acyclic graphs (DAGs). Flink programs typically apply transformations on data-sources and save the results to data-sinks before exiting. Flink has the special classes DataSet for bounded datasets, and DataStream for unbounded data-streams, to represent data in a program. To summarise, Flink programs look like regular programs that transform data collections. Each program consists of: (i) initialising the execution environment, (ii) loading datasets, (iii) applying transformations, (iv) specifying where to save the results. Flink programs use a lazy execution model, i.e. when the programs main method is executed, the data loading and transformations do not happen immediately. Rather, each operation is added to the program's plan, which is executed when its output needs to be used immediately. This contrasts with a flow-based programming model of mashup tools, which relies on an eager evaluation model i.e., a flow component is first executed before the control flows to the next component. This difference must be taken into consideration while enabling Flink programming from mashup tools.

Design Decisions

In order to support Flink pipelines in mashup tools, we needed to decide on the (i) required abstraction level, (ii) the execution model mapping and (iii) the way to support semantic validity of graphical flows. Accordingly, from the different abstraction levels, we decided to select the core API abstraction levels for supporting Flink pipelines in graphical mashup tools, as these APIs are easy to represent in a flow-based programming model. They prevent the need for user-defined functions to bring about data transformation 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 DataStream and DataSet 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. We follow the lazy execution model while composing a Flink pipeline graphically, i.e., when a user connects different components, we do not automatically generate Flink code but instead take a note of the structure and capture it via a DAG, simultaneously checking for semantic validity of the flow. When the flow is marked as complete, the runnable Flink code is generated. Lastly, we impose semantic validity restrictions on the graphical flow which can be composed by the user, i.e. it must begin with a data-source component, followed by a set of transformation components and finally ending with a data-sink component, in accordance with the anatomy of a Flink program.

4 Designing Flink pipelines

The conceptual approach for designing Flink pipelines via graphical flows addresses the main contributions stated in Section 1, and consists of: (i) A model to enable the graphical creation of programs for stream analytics, in other words, to automatically translate items specified via a GUI to runnable source code, known as the *Translation & Code Generation Model*, and (ii) a model to continuously assess the end-user flow composition for semantic validity and provide feedback to ensure that the final graphical flow yields a compilable source code, known as the *Validation Model*. Figure 3 gives a high-level overview of the conceptual approach used to achieve such a purpose. This conceptual approach is based on the design decisions discussed in Section 3.

Since the main idea is to support stream analytics in mashup tools, we restrict the scope of the translator to the DataFrame APIs from the core-level API abstractions. In accordance to the anatomy of a Flink program, we have built *'SmartSantander Data'* as the data-source component, an *'Output Result'* supporting writing operation to Kafka, CSV and plain text as data-sink component. Map, filter and window operations are the supported transformation components. Accordingly, we built the *'GPS Filter'* component to specify filter operations, the *'select'* component to support map operations and a *'Window'* as well as *"WindowOperation"* to specify windows on data streams. We also support the Flink CEP library via the following components :*'CEP Begin'*, *'CEP End'*, *'CEP Add condition'* and *'CEP New condition'*. The CEP library is used to detect patterns in data streams. We also have two additional components, namely *'Begin Job'* and *'End Job'*, to mark the start and end of a Flink pipeline. The translator & code generation model have been designed to work within this scope of selection. We define all potential semantic rules between these components and the validation model works within this scope.

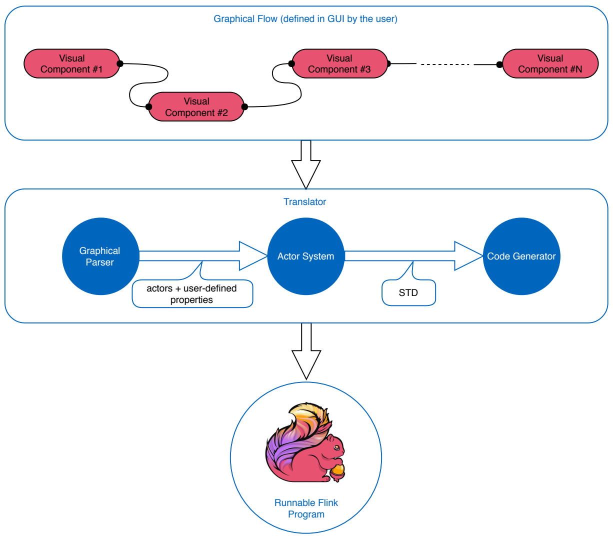


Fig. 3: Conceptual Approach for Translation and Code Generation

4.1 Translation & Code Generation

The aim of the translation & code generation model is to provide a way to translate a graphical flow defined by the end user of the mashup tool (via its GUI), into source code to program Flink. This model behaves as follows: (i) First, end users define graphical flows in the mashup tool GUI, by connecting a set of *visual components* in a flow-like structure. It represents a certain Flink functionality and has a set of properties that the user may configure according to their needs. (ii) Then, a *translator* acquires the aggregated information of the user-defined flow, which contains (a) the set of visual components that compose the flow, (b) the way in which they are connected, (c) the properties that users have configured for each component.

The *translator* has three basic components: a *graphical parser*, an *actor system* and a *code generator*. It takes as input the aggregated information of the user-defined graphical flow (i.e. visual components, the flow structure and the user-defined properties) and its output is a packaged and runnable Flink job. The graphical parser takes the aforementioned aggregated information and processes it, creating an internal model and instantiates the set of actors corresponding to the flow. The actor system is the execution environment of actors, which contains the business logic of the translator. Actors are taken from the output of the graphical parser. The actor model abstraction makes each actor independent, and the only way to interact with the rest is by means of exchanging messages. Actors communicate using a data structure that has been explicitly defined for making the translation, using a tree-like structure that makes appending new nodes extremely easy. In this model, the data structure is referred to as STDS (Specific Tree-Like Data Structure). As previously stated, each actor corresponds to a specific Flink functionality and, in turn, to the standalone implementation method of that specific functionality. It adds a generic method-invocation statement as a message response to the next connected actor. The method-invocation statement also passes the user parameters and the output from its preceding node as input to the standalone implementation method of Flink-functionality APIs. The next actor receives this message and appends its corresponding method-invocation statement and so forth.

Finally, the code generator takes the STDS as input. It has internal mapping to translate parametrised statements into real Flink source code statements. This entity combines the parametrised statement with this mapping and the user-defined properties, and then generates the final source code. The compiling process also takes place here. The code generator output is a packaged, running Flink job that can be deployed in an instance of Flink.

4.2 Validation

The translation model allows the translation of graphical flows into source code. However, some graphical flows may result in source code that either cannot be compiled or yields runtime errors. We have provided support on aFlux for handling the type of errors that occur because of data dependencies in a data pipeline, during the specification of the pipeline from the GUI. If one of the data dependency rules is violated when the user connects or disconnects a component in a flow, visual feedback is provided, which helps avoid problems early on. Such semantic rules must be specified by the developers of the individual Flink components of aFlux, according to the following pattern:

$$\underbrace{\text{Component A}}_{\substack{\text{main}\\ \text{visual component}}}\ \underbrace{\begin{vmatrix}\text{should}\\ \text{must}\end{vmatrix}}_{\text{isMandatory}}\ \text{come}\ \underbrace{\text{(immediately)}}_{\text{isConsecutive}}\ \underbrace{\begin{vmatrix}\text{before}\\ \text{after}\end{vmatrix}}_{\text{isPrecedent}}\ \underbrace{\text{Component B}}_{\substack{\text{argument}\\ \text{visual component}}}$$

For example, the following rules can be specified:

– *'Window' component must come immediately after 'Select' component*
– *'End Job' component must come after 'Load data' component*

On the front-end, when a user connects two components, it is considered a state-change. With every state-change, the entire flow is captured from the front-end and subjected to the validation process. Basically, the flow is captured in the form of a tree; the next step is to check whether the nodes are compatible to accept the input received from their preceding nodes, whether two immediate connections are legal and whether the tested component's positional rules permit it to be used after its immediate predecessor. Algorithm 1 summarises the semantic validation steps of the flow. During the check, if an error is found with any one component of the flow, the user is alerted with the appropriate reasons and the component is highlighted.

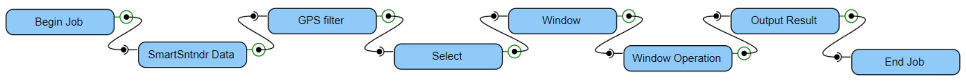
Algorithm 1: Continuous Semantic Validation of Flink Pipelines

```
foreach flow in the canvas do
    order the list of element as they appear in the flow;
    foreach element in the orderedList do
        instantiate the PropertyContainer that corresponds to element;
        get the set of conditions out of it;
        instantiate a new result;
        foreach condition in conditions do
            foreach element in the orderedList do
                if condition is not met then
                    result.add(condition);
                end
            end
        end
        if result is empty then
            clear error information from element;
        else
            add error information to element;
        end
    end
end
```

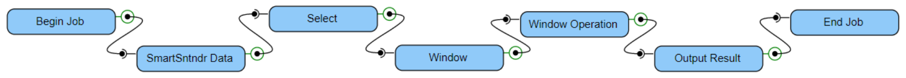
5 Evaluation and Discussion

The implemented approach has been evaluated for its ease in graphically creating Flink jobs from aFlux and abstracting the code-generation from the end-user. For evaluation purposes, we have used real data from the city of Sandander, Spain, which is offered as open data behind public APIs [14]. In this smart city use-case, the user is an analyst of Santander City Hall, who need not have programming skills. The user only needs to know how to use aFlux from the end-user perspective (e.g. drag and drop mashup components) and have some very basic knowledge of what Flink can do from a functionality point of view rather than from a developer point of view. For example, the city hall analyst should know that changes in the city are measured in events and events can be processed in groups called windows. The user does not need to know any details about how to create a window in the Flink Java or Scala API, or the fact that generics need to be used when defining the window type of window. The process of analysing real-time data involves combining data from different sources of the city and processing it. The goal of this use-case is to gain insights about the city, that help decision makers take the appropriate calls.

In this evaluation scenario, temperature vs. air quality in a certain area must be compared with the average of the city. To study the relationship between the level of a certain gas and temperature, the analyst needs to create four flows (or wire them all together to create a simple Flink job): two of them will analyse temperature data (i.e. the 'temperature' attribute in the 'environment' dataset) and two of them will analyse air quality (e.g. the 'levelOfCO' attribute in the 'airQuality' dataset). Two flows are required for each dataset because one will include a 'GPS filter component (Fig. 4a), and the other one will not include it, in order to process all the data in the city (Fig. 4b). To avoid re-adding the same mashup components, the analyst could make use of the *sub-flow* feature of aFlux.



(a) Flow A

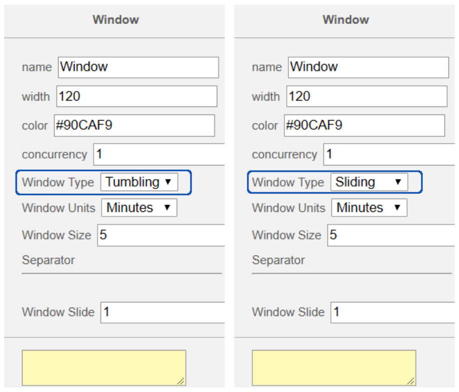


(b) Flow B

Fig. 4: Flows in aFlux for Case Study - Real-Time Data Analytics

Figure 4 shows how the analyst can easily get input from real-time sources by using a graphical data-source component, i.e. the 'SmartSntndr Data'. Adding a third source of data to see not only the level of NO_2 but also the level of ozone is as simple as changing a property in the 'SmartSntndr Data' component. However, if they were doing it manually, the Java code for a new 'MapFunction' would have to be written.

Tumbling windows were used in Figure 4a, but processing the data in a different type of window (e.g. using sliding windows) is as easy as changing the properties of the 'Window' mashup component (Figure 5). In Java, the user



	Window	Window
name	Window	Window
width	120	120
color	#90CAF9	#90CAF9
concurrency	1	1
Window Type	Tumbling	Sliding
Window Units	Minutes	Minutes
Window Size	5	5
Separator		
Window Slide	1	1

Fig. 5: Tumbling vs. Sliding Windows in aFlux

would need to know that a sliding window takes an extra parameter and that the window slide needs to be specified using Flink's Time class, in which a different method is invoked depending on the desired time units.The system has been evaluated against additional scenarios and case studies [13].

Discussion

The approach used to model a Flink pipeline relies on three aspects, i.e. load data from data source, transform data and finally publish the result via a data sink. This is also the preliminary form of semantic validation i.e. deciding if the positional hierarchy of a component is allowed or not. The user-flow is parsed and expressed as an abstract syntax tree which is passed as an input to the code generator. Each node in the tree maps to a standalone implementation of the Flink Core APIs. The code generator generates code for sequences like, opening and closing a Flink session, and for the nodes in the abstract syntax tree it wires the standalone implementation of the APIs, while passing the user parameters and the output from the preceding node as input. The result is a runnable Flink program, compiled, packaged and deployed on a cluster.

The current work has many limitations in its approach such as the following.

Debugging run-time exceptions: The semantic validation techniques described help the user create a flow which can result in a compilable Flink code. Nevertheless, in the case of run-time exceptions, it becomes difficult to identify the error from the logs and reverse map the generated Flink program to identify the corresponding graphical component on the front-end.

Integrate job monitoring: The current approach does not include methods to include job monitoring and management features at the tool level. The user can create a Flink job and consume the analytical result, but the user cannot manage the job deployed on a Flink cluster. This is important from an end-user perspective as stream applications typically run infinitely.

Seamless integration with Flink cluster: Currently, there is no seamless integration between the mashup tool and Flink run-time environment, hence the consumption of the data analytics results is not a straightforward process. Therefore, real-time data visualisation has many problems, including time-delays and unresponsiveness to very minimal interactive capabilities. As of now, we rely on third party systems, like Apache Kafka, where the Flink application writes its results to, and we read the data from, Kafka in the mashup tool.

6 Related Work

We did not find any mashup tool solutions which allow wiring components to produce a Flink application. One of the closest existing solutions is Nussknacker [3]. It is a tool currently in development which supports graphical Flink programming. It consists of an engine, whose aim is to transform the graphical model created in the GUI, into a Flink job. A standalone user interface application, which allows both the development and deployment of Flink jobs, is written in Scala and incorporates data persistence and a Flink client. Basically, a user needs to enter the data model of their use-case to Nussknacker. Users with no programming skills can benefit from the GUI to design a Flink job, send it to a Flink cluster and monitor its execution. Nevertheless, it does not focus on the integration of data analytics and business logic of application, but rather designs a data analytics Flink application based on a particular usage model. IBM SPSS Modeller provides a GUI to develop data analytics flows involving simple statistical algorithms, machine learning algorithms, data validation algorithms and visualisation types [2]. 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 Flink applications.

7 Conclusion

We defined a new approach for high-level Flink programming from graphical mashup tools to make the usage of stream analytics easier for non-domain experts. We showed that this is feasible and evaluated what are the right abstractions. Accordingly, (i) we analysed the Flink ecosystem i.e. its distributed data-flow programming model and the various abstraction levels offered to program applications; we found the core APIs, based on DataFrame and DataSet interfaces to be the most suitable candidates for use in a graphical flow-based programming paradigm, i.e. mashup tools; (ii) we adapted the eager evaluation execution model of mashup tools to support designing Flink pipelines in a lazy fashion and devised a novel generic approach for programming Flink from graphical flows. The conceptual approach was implemented in aFlux, our JVM actor-model-based mashup tool and evaluated it with real-time data from the city of Santander.

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. IBM Node-RED, A visual tool for wiring the Internet of things, http://nodered.org/
2. IBM SPSS Modeller. https://www.ibm.com/products/spss-modeler, [Online; accessed 22-June-2018]
3. Nussknacker. https://github.com/TouK/nussknacker, [Online; accessed 22-September-2018]
4. Atzori, L., Iera, A., Morabito, G.: The internet of things: A survey. Computer Networks **54**(15), 2787 – 2805 (2010). https://doi.org/http://dx.doi.org/10.1016/j.comnet.2010.05.010, http://www.sciencedirect.com/science/article/pii/S1389128610001568
5. Daniel, F., Matera, M.: Mashups: Concepts, Models and Architectures. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)
6. Friedman, E., Tzoumas, K.: Introduction to Apache Flink. O'Reilly (09 2016)
7. Health, N.: How ibm's node-red is hacking together the internet of things (March 2014), http://www.techrepublic.com/article/node-red/TechRepublic.com [Online; posted 13-March-2014]
8. Iqbal, M.H., Soomro, T.R.: Big data analysis: Apache storm perspective. International journal of computer trends and technology **19**(1), 9–14 (2015)
9. Kiran, M., Murphy, P., Monga, I., Dugan, J., Baveja, S.S.: Lambda architecture for cost-effective batch and speed big data processing. In: Big Data (Big Data), 2015 IEEE International Conference on. pp. 2785–2792. IEEE (2015)
10. Mahapatra, T., Prehofer, C., Gerostathopoulos, I., Varsamidakis, I.: Stream analytics in iot mashup tools. In: 2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). pp. 227–231 (Oct 2018). https://doi.org/10.1109/VLHCC.2018.8506548
11. Mahapatra, T., Gerostathopoulos, I., Prehofer, C., Gore, S.G.: Graphical spark programming in iot mashup tool. In: The Fifth International Conference on Internet of Things: Systems, Management and Security. p. In Press. IoTSMS (2018)
12. Marz, N., Warren, J.: Big Data: Principles and best practices of scalable real-time data systems. New York; Manning Publications Co. (2015)
13. Moreno, F.A.F.: Modularizing flink programs to enable stream analytics in iot mashup tools (2018), http://oa.upm.es/52898/
14. Santander City Council: Santander Open Data - REST API Documentation (2018), http://datos.santander.es/documentacion-api/
15. Stonebraker, M., Çetintemel, U., Zdonik, S.: The 8 requirements of real-time stream processing. ACM Sigmod Record **34**(4), 42–47 (2005)
16. The Apache Software Foundation: Dataflow Programming Model, v1.5 (2018), https://ci.apache.org/projects/flink/flink-docs-release-1.5/concepts/programming-model.html