VTDL: A Notation for Data Stream Processing Applications

Christoph Hochreiner,
Stefan Schulte and Schahram Dustdar
Distributed Systems Group,
TU Wien, Austria
{c.hochreiner, s.schulte, s.dustdar}
@dsg.tuwien.ac.at

Matteo Nardelli
University of Rome
Tor Vergata, Italy
nardelli@ing.uniroma2.it

Bernhard Knasmueller
TU Wien, Austria
bknasmueller@gmail.com

***Abstract*—The continuing growth of the Internet of Things (IoT) requires established stream processing engines (SPEs) to cope with new challenges, like the geographic distribution of IoT sensors and clouds hosting the SPEs. These challenges obligate SPEs to support distributed stream processing across different geographic locations which also require a new approach on how data stream processing topologies are defined. In this paper, we identify required features for next-generation SPEs and introduce the Vienna Topology Description Language (VTDL). This language is specifically designed to address challenges for next-generation SPEs and proposes several novel aspects compared to existing topology description concepts. To assess not only the feasibility but also the reduced management overhead due to the VTDL, we evaluate the VTDL within the VISP stream processing ecosystem and show that the usage of the VTDL approach results in a management time reduction of up to 18 times.**

I. Introduction

Due to the constant rise of the Internet of Things (IoT) and the transition towards a data-centric society, today's stream processing engines (SPEs) need to deal with a continuous increase of streaming data concerning volume, variety, and velocity [1]. These three properties are the main drivers for the evolution of SPEs like System S [2], Apache Storm [3] or Apache Spark [4] as well as the design of new ones like Heron [5] or CSA [6]. These SPEs are designed to run on clouds and excel at processing huge data volumes at high velocity for social media applications, like Twitter [5] and LinkedIn [7] or other areas such as financial analysis [2]. However, they do not consider the geographic location of IoT sensors and of cloud-based computational resources used to run the SPEs [8].

The structure of stream processing applications (SPAs) is defined by *topologies*, which represent a choreography of stream processing operators which manipulate the data [2] provided by data sources. Most data sources in the IoT, e.g., sensors, are running in different geographic locations which are often far away from computational resources on public clouds [6]. This data needs to be transferred to SPEs which can either be directly running on private clouds next to the data sources or on rather centralized public clouds over the Internet which requires networking resources [9]. After the data is processed, it needs to be delivered to the designated receiver, which is either in proximity to the data source, e.g., actuators for the data source, or at an arbitrary geographic location, e.g., a system operator who wants to monitor the status of manufacturing machines on a mobile device.

Up to now, most SPEs either deploy multiple SPE instances close to the data sources on private clouds or deploy one centralized SPE on a public cloud [6]. Both approaches exhibit major problems: The distributed approach promises low latency but requires high setup efforts by establishing the communication among the different SPEs. The centralized approach exposes a high degree of latency as well as potential network congestions because the data needs to be transferred to the SPE over the network before it can be processed [10].

Furthermore, today's SPAs are often subject to change [11]. The changes are mostly triggered by operational aspects at run time, e.g., due to reconfigurations of operators. These reconfigurations can occur due to Quality of Service (QoS)-related aspects, like changes in the underlying computational infrastructure, e.g., addition or removal of computational resources [12], or due to changes in the incoming workload which require the replication of specific operators [13].

Besides these operational changes, there are also other reasons to update the deployment of SPAs. These reasons range from updates for single operators, e.g., to fix software bugs, to organizational changes, like the addition of new data sources and consumers, e.g., sensors and users, or new legal restrictions that enforce the processing of data within a distinct geographic area [14]. To improve the management of SPAs in terms of their management overhead and deployment time especially across different geographic locations, we propose the Vienna Topology Description Language (VTDL), which extends on the concepts of SPL [15] which was introduced for IBM System S and supports the outlined features. Furthermore, we have integrated the VTDL into the VISP Ecosystem[1] to provide a reference implementation for the VTDL. In addition, we have designed a protocol which allows the use of the features enabled by the VTDL, such as the isolated operator updates while continuing the data processing in the unaffected part of the SPA.

[1]Available as open source software at https://visp-streaming.github.io

0-7695-6394-5/18/$31.00 ©2018 IEEE
DOI 10.1109/SOSE.2018.00019

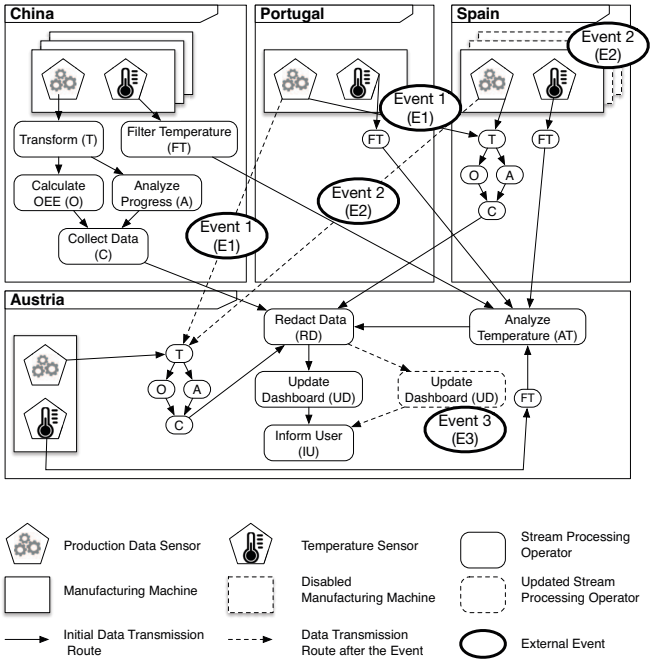

Fig. 1: Motivational Scenario

The remainder of this paper is structured as follows: First, we provide a motivational scenario in Section II and discuss the required features for next-generation SPEs in Section III. In Section IV, we introduce VTDL and in Section V we discuss the requirements for SPEs to support the VTDL. Section VI presents the evaluation of the VTDL and a discussion thereof. Section VII discusses the related work and Section VIII concludes the paper.

II. Motivation

A. Topology Structure

In this section, we provide a simplified stream processing topology from the manufacturing domain, which is inspired by the EU H2020 project CREMA (Cloud-based Rapid Elastic Manufacturing) [16]. For our scenario, we consider a company with production facilities in China, Portugal, Spain, and Austria as well as a headquarter in Austria as shown in Figure 1. Each geographic location hosts a different amount of manufacturing machines which are equipped with sensors. In our scenario, we consider two sensors per manufacturing machine as data sources, and several operators that process data before presenting the information to users.

The first sensor is the *Production Data Sensor*, which emits production-related attributes from the manufacturing machines every minute, such as produced items, faulty items, or downtimes. This information is encoded in a binary data format and needs to be transformed into machine-readable data before it can be processed by succeeding operators. The transformation is conducted by the Transform (T) operator that also forwards the data to succeeding operators. These operators calculate the Overall Equipment Efficiency (OEE), a metric commonly used in the manufacturing domain to assess the usage rate of their manufacturing machines [17], and analyze the production progress. These metrics are then collected by the Collect Data (C) operator to be transferred to the central reporting location, i.e., the headquarter in Austria. Here, the Redact Data (RD) operator redacts the individual metrics and triggers the Update Dashboard (UD) operator, which shows relevant information on a dashboard to inform the user about the current efficiency and the overall progress.

The second sensor is the *Temperature Sensor*, which emits temperature readings of the machine every second. These temperature readings are filtered by the Filter Temperature (FT) operator to only forward distinct temperature readings, e.g., when the temperature exceeds a predefined threshold. However, due to the high volume of data, it is imperative to locate this filtering operator near the sensor to avoid a large volume of data to be transferred over the Internet. After the filtering operation, the critical readings are forwarded to an Analyze Temperature (AT) operator before they are presented to the user.

B. Deployment Scenarios

Although the topology is identical for all four production locations, there are different deployment possibilities, as shown in Figure 1.

The first deployment scenario is to split the topology into two parts and deploy all metric calculation operators near the data source, e.g., on a private cloud, as done for the plant in China. Due to the high amount of data from the manufacturing machines, it is reasonable to preprocess the data next to the data source and only transfer the filtered data over the Internet.

For the second deployment scenario, we consider the plant in Portugal, which only hosts the filter operator locally and forwards all other data to a private cloud located at the plant in Spain. This common usage of computational resources requires only little computational resources for the plant in Portugal, relatively low network cost due to the small number of machines, and low geographic distance as well as a higher utilization of the private cloud in Spain. After calculating the metrics on close to the data sources, the results are then sent to the headquarter in Austria for the remainder of the SPA.

Finally, the last deployment scenario can be found for the plant and headquarter in Austria. For this scenario, all operators are located on a big private cloud of the plant in Austria. This geographic co-location avoids the transfer of the data from the manufacturing machines over the Internet.

C. Topology Changes

While the topology in Figure 1 may appear rather fixed at first, there are several events which can occur at run time.

The first event (E1 in Figure 1) is a communication outage. In our scenario, the manufacturing machines in Portugal and Spain use the same computational infrastructure in Spain to calculate the OEE and progress metrics.

This setup is feasible as long as the network connection between these two plants is intact. Nevertheless, whenever there is a communication outage, the SPA needs to be reconfigured to continue data processing. Based on the geographical location, it is possible to reroute the raw data to Austria to compensate the outtake and resume data processing.

The second event (E2) represents a volume reduction for the SPE. Occasionally, manufacturing machines have downtimes and the full data processing capacities of the SPE are not required anymore. Whenever the full processing capabilities are not required anymore, it is feasible to release computational cloud resources and reroute the data to reduce the total operational cost. For E2, several manufacturing machines are switched off in Spain and similar to the previous event it is possible to reroute the raw production data to Austria for processing.

While the first two events are triggered by operational aspects, it may also be required to replace individual operators due to organizational reasons. For the third event (E3), we consider a software update for the UD operator. This software update fixes an internal flaw of the operator implementation, but the overall semantics of the operator, i.e., the input and output data structure, remains the same. For this update, it should not be required to redeploy the whole topology, as required for most of the established SPEs, like Apache Storm or Apache Spark. SPEs should only need to buffer the incoming traffic for a short time until the new UD operator is in place and can continue with its operations.

It should be noted that the simplified example provided in this section is illustrative only. In fact, the handling of data streams is not a singularity in the smart manufacturing, but rather a prerequisite in smart systems in general, e.g., smart cities [18], smart grids [19], or smart healthcare [20].

III. Features for Next-generation SPEs

Based on the motivational scenario, we identify some basic as well as six next-generation features that are not available in today's SPEs (see Section VII). The main difference between the basic features and the next-generation features is that the basic features are already mostly covered by existing topology description languages, such as SPL [15] or CQL [21].

A. Basic Features

The primary feature of any topology description approach is to define how data sources, operators, and information consumers, are connected to realize an SPA. Stream processing topologies are usually represented as directed acyclic graphs [15], where vertices represent the operators and edges represent the data streams between the operators. A data source feeds data into the topology and has no incoming data streams. Hence, each topology requires at least one data source, but there can be arbitrarily many. The data provided by the sources is then processed by one or more operators.

Operators execute user-defined code, whether it is a simple operation like filtering, aggregation, merging or more complex ones such as regression, classification [2]. Operators can obtain data from arbitrarily many vertices, i.e., data sources or other operators, and emit new data to other vertices, i.e., other operators or data sinks. Data sinks or consumers represent the endpoints of an SPA since they only consume data, i.e., only have incoming edges and each SPA needs at least one data sink.

B. Next-Generation Features

Deployment Preferences (F1): The most important feature for geographically distributed SPAs is deployment preferences for individual operators [6]. While this is not relevant for SPAs in a single location, it becomes crucial for geographically distributed ones. Each operator needs to be able to provide a set of admissible deployment locations where it can operate and satisfy real-world constraints. These constraints mainly affect data sources, e.g., a temperature sensor or a camera, which are running in a fixed location cannot be relocated. Additionally, it may also be prohibited to transfer specific data, e.g., medical data, to certain locations like public clouds [14] which also limits the deployment of some operators.

QoS Compliance (F2): Although QoS compliance is commonly used for software services, such as SPAs [22], it is, to the best of our knowledge, not considered by state-of-the-art SPEs on an operator level.

While the application-level QoS compliance may be sufficient for most users, research for microservices has shown that a more fine-grained approach, i.e., on an operator level, allows identifying bottlenecks. Based on such a bottleneck analysis resource provisioning algorithms can achieve lower costs by only scaling specific operators instead of the whole SPA [23]. Furthermore, an operator-level QoS compliance also allows the integration of external operators on a software-as-a-service basis into SPAs [24].

Fault Tolerance (F3): To compensate operator failures or hardware failures, it is imperative that SPEs are capable of applying automatic failure compensation mechanisms. Nowadays, SPEs already provide basic fault tolerance mechanisms, like the automatic restart of operators whenever they fail [25]. Nevertheless, it is required to also support more sophisticated fault tolerance mechanisms, like deploying an updated topology for SPAs to reroute the traffic (as required for event E1). This feature builds on top of the QoS compliance feature (F2), which allows the SPE to detect operator failures, e.g., based on a high latency, or infrastructure outages.

Operator Composability (F4): To ensure the compatibility among the operators, it is required to provide simple semantic annotations, regarding incoming as well as outgoing data types as already proposed for the streaming data itself [26]. These semantic annotations can be used to check whether operators are compatible and in a further step also to apply an automatic semantic operator selection. This would allow the user to only provide an abstract description for the operator task, and the SPE can autonomously create the topology which reduces the users' workload for designing SPAs. This feature is already available for other domains like sensor networks [27], but is still missing for SPAs running on SPEs.

Topology Modifications at Runtime (F5): The need for topology modifications at run time, as required for the third compensation mechanism (E3) in the motivational scenario, has also been identified in the literature [28], [29]. Stream processing topologies are often deployed for long-term data processing, which makes it hard to apply minimal updates, such as bug fixes for individual operators, without redeploying the whole SPE. Therefore, the SPE needs the possibility to pause the data flow for individual operators, to apply the update. As long as the operator composability (F4) does not render any inconsistencies, it is sufficient to only pause the processing for the operator that needs to be updated. This allows to update topologies with software updates or enable fine-grained failure compensation measures.

Different Data Transfer Modes (F6): Due to the geographically distributed deployment, it is required that some operators are connected via the Internet instead of a local connection which is common for centralized SPE deployments. The network connection over the Internet may result in a high communication overhead because each data item is sent individually which is only efficient in local settings. To mitigate this communication overhead, it is often more efficient to create so-called micro-batches, i.e., to aggregate multiple data items and send them as a single group (or batch) over the Internet. This reduces not only the communication overhead and network load but also improves the performance of the data processing [30].

IV. VTDL – Vienna Topology Description Language

The goal of the VTDL is to support both the basic features, as already present for existing topology description approaches, as well as the next-generation features that we have identified in the previous section. We have chosen the SPL [15], which was initially developed for System S as a starting point since it already provides some of the identified features compared to other description approaches (see Section VII). The description of the SPA topology is provided through a VTDL file, which contains a list of the operators, their roles, and attributes, as well as information on how they are connected.

In VTDL, each vertex of a topology is identified by a textual identifier, which is prefixed with a $ character, as presented in Listing 1. Directly after the identifier, the role of the vertex is indicated, namely *Source()*, *Operator()*, or *Sink()*; the latter two roles need to take at least one operator identifier as the input parameter, to describe the data flows between the operators. When the operator needs to receive data from several upcoming sources, their identifiers are specified in a comma-separated list, e.g., *$oeeS, $progressS*. In addition to the structural description, each vertex is assigned a set of key-value pairs, as listed in Table I, which include attributes of interest for the deployment time and run time management. These key-value pairs can be categorized into three categories: required attributes, attributes with default values, and optional attributes.

Listing 1: Excerpt of a Topology Description for an SPA

```
$productionData     = Source() {
  concreteLocation : ":::::ffff:8083:c001/cpu",
  type             : "source",
  outputFormat     : "productiondata"
}
$transform          = Operator($productionData) {
  allowedLocations : ":::::ffff:8083:c001"
                     ":::::ffff:8083:c002",
  poolPreferences  : "cpu gpu",
  concreteLocation : ":::::ffff:8083:c001/cpu",
  inputFormat      : "productiondata",
  type             : "transformData",
  outputFormat     : "machinereadableProductionData",
  stateful         : "false",
  maxResponseTime  : "0.5"
}
$oee                = Operator($transform) {
  allowedLocations : ":::::ffff:8083:c001"
                     ":::::ffff:8083:c002",
  inputFormat      : "machinereadableProductionData",
  type             : "oeeCalculator",
  outputFormat     : "oeeData",
  compensation     : "redeployTopology"
}
$progress           = Operator($transform) {
  allowedLocations : ":::::ffff:8083:c001"
                     ":::::ffff:8083:c002",
  inputFormat      : "machinereadableProductionData",
  type             : "analyzeProgress",
  outputFormat     : "progressData",
  compensation     : "mailto:admin@tuwien.ac.at"
}
$collectData        = Operator($oee, $progress) {
  allowedLocations : *,
  poolPreferences  : "ssd",
  concreteLocation : ":::::ffff:8083:c001/ssd",
  inputFormat      : "oeeData progressData",
  type             : "collectData",
  outputFormat     : "dataCollection",
  maxQueueLength   : "200",
  protocol         : "microbatch/50items",
  compensation     : "deploy:www.visp.io/backup.vtdl"
}
$informUser         = Sink($collectData) {
  concreteLocation : ":::::ffff:8083:c002/general",
  inputFormat      : "dataCollection",
  type             : "informUser"
}
```

The most important attribute is the *type* of the vertex, which describes the functionality of the operator, whose concrete implementation is resolved by the SPE. The second most important attribute of the VTDL is the location aspect of the operator as required for F1. Each location is identified by an IP address, which defines the concrete location of the SPE, data source or data sink and is a must-have attribute. The default value is *, which does not restrict the deployment locations. Nevertheless, it is also possible to restrict the locations by providing a list of allowed ones.

Each geographic location may have different resource types available, e.g., resources with specific hardware aspects, like solid state drives (SSDs), high-performance CPUs or GPUs. VTDL assumes that these resources are handled as resource pools, e.g., a resource pool with high-performance CPUs, and the SPA designer can indicate deployment preferences by providing a list of *poolPreferences*. This allows the SPE deploying the operators according to their resource preferences if the specific hardware is available. If the specific hardware is not available, the SPE will select any computational resource which is available.

TABLE I: Operator Attributes

Attribute	Mandatory	Values	Default	Description
type	✓	string	–	Reference to the operator logic
allowedLocations	✓	IP+	–	Allowed deployment locations
poolPreferences		string+	general	Hardware preferences
concreteLocation		IP/poolIdentifier*	–	Preselected location
inputFormat		string+	–	Implemented input formats
outputFormat		string	–	Data output format
maxResponseTime		numerical	5s	Maximum operator response time
maxQueueLength		numerical	100	Maximum queued messages
stateful		{true | false}	true	Presence of operator state
replicationAllowed		{true | false}	false	Operator replication
protocol		{stream | microbatch:<X>items | microbatch:<X>ms }	stream	Operator processing mode
compensation		{redeploySingle | redeployTopology | deploy:<URL> | mailto:<email> | none }	redeploySingle	Failure recovery mode

The *concreteLocations* value is composed of an IP address and the pool identifier. This attribute needs to be provided at design time for data sources and sinks since they are fixed. For operators, this attribute is optional, but it can be used to indicate concrete deployment instructions. All other concrete locations are selected at deployment time, as discussed in Section V-B. Later at run time, this concrete location can be updated if necessary to improve the performance of the SPA, e.g., to recover from a failure or to meet QoS requirements.

Besides the type attribute, the VTDL also features several attributes for the semantic description of the operator to enable composability checks (F4) or automatic semantic operator selections. Due to the fact that these composability checks are not essential for the data processing, these attributes are optional. These composability checks are represented by the *outputFormat* attribute that defines the semantic type of the data, which is forwarded to succeeding vertices for each source and operator and the counterpart (*inputFormat*), which is used for all operators and sinks.

The next category of attributes considers the QoS aspects of the operator (F2). Up to now, the VTDL considers the *maxResponseTime* for one processing operation and the *maxQueueLength*, which measures the amount of data items waiting for processing. Nevertheless, there are also other QoS aspects like the maximum CPU or memory utilization for a particular operator, which could be easily added as attributes for the operators. To inform the execution framework regarding the operator behavior at run time, VTDL comprises three more attributes: stateful, replicationAllowed, and protocol.

The *stateful* attribute describes whether the operator is stateful, i.e., computes the output data using the incoming data together with internal state information. This attribute is required to indicate the required effort to migrate operators, because it is easy to relocate stateless operators, but it requires extra effort to migrate the state for stateful ones [31].

The *replicationAllowed* attribute describes whether multiple instances of the same operator can be executed concurrently by the SPE, whereas the SPE needs to take care of the concrete partitioning-scheme of the data. Both operational attributes *stateful* and *replicationAllowed* are optional. To apply a conservative approach to preserve the application integrity, we recommend to assume by default each operator as stateful and with no replication allowed.

The *protocol* attribute (F6) allows for defining the data transmission type (see the *collectDataS* operator in Listing 1). It can take two types: *stream*, which is the default option, and *microbatch*, which requires collecting data in groups (i.e., batches) before applying the operator function. For the latter option, the batch size for the micro-batch is provided by either the number of items for the micro-batch, as shown in Listing 1 or by the time in milliseconds, e.g., 100 ms, for sliding intervals. The default setting for this attribute is the individual transmission and it only needs to be set to enforce the micro-batch transmission.

The final attribute is the *compensation* attribute (F3), which describes the failure compensation mechanism in case one operator becomes unavailable. Currently, VTDL supports four different failure compensation mechanisms, as shown in Listing 1. The first compensation mechanism scope is motivated by the literature [32] and is identified by the keyword *redeploySingle*: it requires to deploy a new instance of the faulty operator, cache the incoming data during the new instance startup time, and finally replay the cached data.

The second compensation mechanism, identified by *redeployTopology*, requires restarting the whole SPA, thus resulting in a possible loss of currently cached and processed data. This data loss is also the case for the third compensation mechanism, which allows to deploy an alternative topology when a failure of the current one occurs (as required for E1); the option *deploy:<URL>* specifies this compensation mechanism, where the URL indicates the location of an alternative topology which is deployed as a replacement for the existing one. The last option two options are strictly speaking not a compensation mechanism. They allow the SPE to notify a user via email, i.e., *mailto:<email-address>*, or to simply ignore the failure indicated by *none*.

V. Management for VTDL

To enact topologies, which are defined based on the VTDL, SPEs are required not only to parse the incoming topology descriptions but also to apply topology modifications at run time (F5). First, we provide a short overview of the VISP architecture, which serves as a reference implementation for the VTDL. Then, we present the required functionality by the VISP Runtime to support all features of the VTDL.

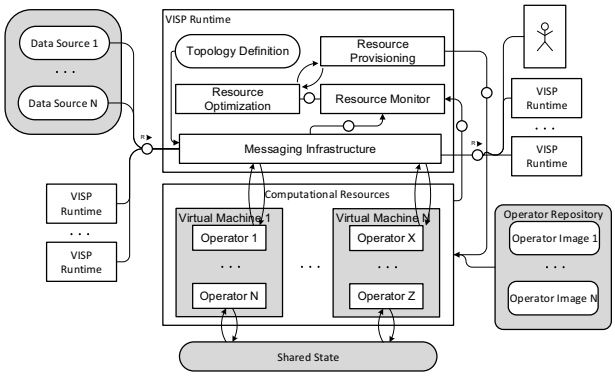

Fig. 2: Architecture of the VISP Ecosystem

A. System Architecture of the VISP Ecosystem

This section provides a short overview of the architecture of the VISP Ecosystem [24], which uses the VTDL for topology descriptions. The VISP Ecosystem is a distributed research prototype, which is capable of running geographically distributed stream processing topologies without any specific interactions required by the user. The core component is the VISP Runtime, represented in Figure 2 (based on the FMC notation [33]), which works in cooperation with several other components, namely the computational resources and the operator registry. The computing resources are needed for executing the operators, whereas the latter is a library of publicly existing operators that can be used to create new data SPAs.

When an SPA is deployed on computational resources, it processes data emitted by data sources (on the left-hand side of Figure 2), to generate high-value information for users, as shown on the right-hand side of Figure 2. The application can be distributed across multiple VISP instances, which are self-contained regarding data processing. This allows the usage of VISP Runtimes as data sources as well as data sinks to realize distributed SPA topologies.

The data to be processed is provided by data sources or preceding VISP Runtimes that push the data to the messaging infrastructure. The actual data processing is conducted by operators, which are running on computational resources, e.g., virtual machines provided by a public or private cloud. At deployment time, each operator is started from a dedicated operator image, e.g., a Docker Image, which is hosted on an external operator repository. As soon as all operators are started, the operators fetch the data from the messaging infrastructure, process it and return the results to the messaging infrastructure. These results are then either fetched from succeeding operators within the same VISP Runtime or forwarded to other VISP Runtimes in different geographic locations.

The remaining components of the VISP Runtime are in charge of ensuring enough processing capabilities, e.g., by replicating individual operators or initiating failure compensation mechanisms, such as notifications or the redeployment of operators. For a detailed description of the individual components, we refer to our previous work [24].

B. Implementation of the VTDL

In this section, we are going to discuss the required functionality of SPEs based on the reference implementation update procedure for the VISP Runtimes as shown in Figure 3. The overall update procedure consists of 13 steps (S1 – S13) which are conducted by a managing VISP Runtime, i.e., the VISP Runtime that first receives the new topology description as well as all other involved VISP Runtimes, which are affected by the topology instantiation or update. Each new topology update operation, including its initial deployment, is triggered by the upload of a new VTDL file to any VISP Runtime (S1). This VISP Runtime is then promoted to be the managing VISP Runtime and checks the composability of the topology (S2) based on semantic annotations provided by the VTDL file. Then, the managing VISP Runtime evaluates if each operator is already assigned a concrete location. If this is not the case, the managing VISP Runtime assigns concrete locations based on the available locations in the topology grounding step (S3); this location is selected based on values of the allowedLocations key. VISP currently supports two grounding approaches, i.e., selecting the first suitable location for a given pool preference for an operator or selecting a random location based on the available ones.

After the preparation phase, the VISP Runtime informs all other involved VISP Runtimes of the update. To apply only one update at a time, the VISP Runtime checks that no other updates are in place. Therefore, the managing VISP Runtime starts a synchronized query, asking for the status of the other VISP Runtimes (S5). If no other updates are in place, it initializes the update (S6): Each of the involved VISP Runtimes is blocked for other updates, and the managing VISP Runtime distributes the updates to all involved VISP Runtimes. The concrete number of updates depends on the actual changes for the SPA. This set of updates is derived by comparing the currently deployed topology against the new one and generating an update command for each change between the topologies. The update sets either consist of creation commands for new operators, deletion commands for operators which are not required anymore or a reconfiguration command for the messaging infrastructure if the data flow is redirected. When there is no topology available, the set of update commands comprises of the whole topology. But in most cases, there are only small changes to existing topologies which result in partial updates.

After all required update commands are distributed to the affected VISP Runtimes, they evaluate whether the update is feasible based on the locally available computational resources. If this is the case, the managing VISP Runtime is informed of the successful update checks; otherwise, an exception is raised (S8). Up to this step (S8), no changes have been applied to any already running operators, which allows a stop of the topology update command without any compensation mechanism required. The first actual changes

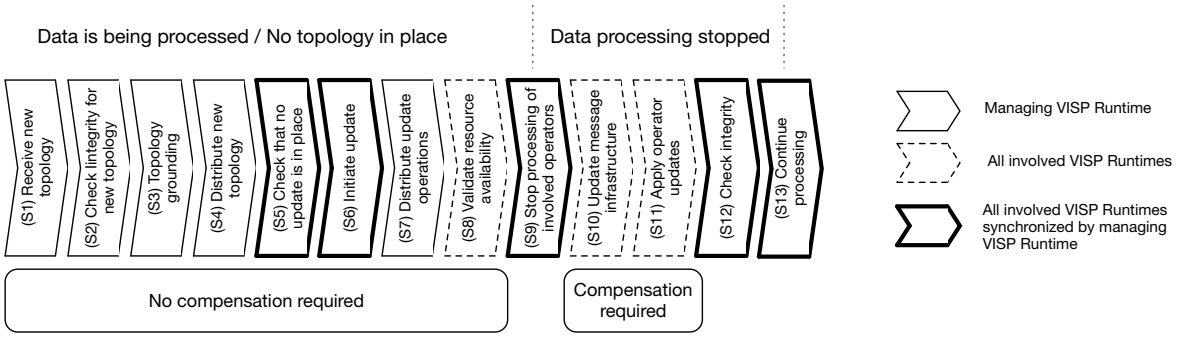

Fig. 3: Topology Update Procedure

are applied in the next step (S9), which triggers a processing stop for all affected operators.

This marks a major distinction in contrast to other SPEs, which need to terminate the complete SPA before deploying new of updated topologies and therefore suffer downtimes. The VTDL approach allows the SPE to continue the data processing for those operators that are not affected by the update operation. After stopping the data processing, each involved VISP Runtime applies the updates to its messaging infrastructure (S10), removes obsolete operators and instantiates new ones (S11). These two steps represent the only critical steps, where manual compensations may be required if any reconfiguration fails. As soon as all update commands have been executed, all VISP Runtimes apply another composability check to ensure that the topology is enacted as intended (S12) and, if this check does not raise any issues, the processing is continued for all operators (S13).

VI. Evaluation

A. Evaluation Scenarios

To evaluate the VTDL and the reduced management overhead for distributing the topology across different geographic locations and applying the topology configuration at each location, we conduct a case study consisting of several scenarios based on the motivational scenario. These scenarios are evaluated regarding duration as well as required user interactions for both the VTDL approach and a baseline approach. The baseline approach represents the state of the art for most established SPEs, e.g., Apache Storm, which do not support any partial updates. In contrast to the VTDL approach, the baseline approach also does not consider SPAs across multiple geographic locations which require the user to apply topology deployment for each geographic location individually.

1. New Topology in one Location: For the first scenario, we assume a topology deployment for a single location. This scenario represents the state of the art for established SPEs and requires no update activities for other SPEs.

2. New Topology across four Locations: The second scenario represents an initial deployment for the motivational scenario. Hereby, the VTDL approach is only required to upload the topology to one VISP Runtime in one location, whereas the baseline approach requires uploading a subset of the topology to all involved locations one after another. The first and the second scenario consider both the messaging infrastructure configuration as well as the operator instantiation for the SPA.

3. Network Disruption (E1): For the network disruption scenario, the VTDL approach can rely on the automatic failure detection and compensation of VISP Runtimes to detect network disruptions between two regions and reconfigure the data flow based on a given alternative topology. This feature is not available for other SPEs and therefore, the evaluation of the baseline approach for this scenario is not possible.

4. Resource Reconfiguration (E2): The resource reconfiguration scenario evaluates the time to reconfigure the data flow between a sensor and an operator for the VTDL approach. This reconfiguration only requires an update to the messaging infrastructure, since the operators are already running in the target location. For the baseline approach, it is required to upload a new topology for all the affected regions, which also requires the deployment of new operators.

5. Single Operator Update (E3): The last scenario evaluates the time required to update a single operator. For this scenario, it is sufficient for the VTDL approach to only update the specific operator, whereas the baseline approach requires the redeployment of the complete topology for the affected location.

B. Evaluation Setup

To conduct the evaluation, we have set up four VISP Runtimes, which represent the individual locations of the motivational scenario, on an OpenStack-based private cloud[2] and on three regions of Amazon EC2[3] to simulate the different geographic regions.

The interactions are conducted by Selenium scripts[4] to eliminate any human-based delays for the evaluation. Each task of these Selenium scripts, e.g., opening a webpage, uploading a VTDL file or removing an enacted topology, is counted as an individual interaction whereas the duration is assessed by the total run time of the Selenium script for the complete scenario.

[2]https://www.openstack.org
[3]https://aws.amazon.com/ec2/
[4]http://www.seleniumhq.org

C. Results and Discussion

To eliminate any potential corruption, e.g., side effects by other cloud users, due to the evaluation in a cloud environment, each scenario was executed three times. Table II shows the average results alongside with the standard deviations of the individual measurements.

For the first scenario, there is no difference between the VTDL approach and the baseline approach, because both approaches follow the same instructions. Each topology update requires three interactions: opening the web-based user interfaces, selecting the desired VTDL file in a file chooser, and initiating the update procedure by clicking on a button. The overall scenario takes about 55 seconds, which is mainly due to the instantiation of Docker Container for each operator. The Docker Images are already available for all scenarios to avoid any network-related delays for downloading the Docker Images.

The first difference between the VTDL approach and the baseline approach can be seen for the second scenario, where the topology is deployed across four locations. The VISP Runtime is capable of deploying the topology to multiple locations in parallel which results in a shorter duration compared to the first scenario, although the update instructions need to be propagated over the network. For the baseline approach, it is required to upload the individual parts of the topology one after another to the individual SPEs to ensure the correct data flow wiring among the different regions. This sequential approach requires significantly more interactions, i.e., four times as much than for the VTDL approach which results in an about 40% faster topology instantiation. Hereby, the majority of the deployment time can also be attributed to the startup duration of the Docker Container.

The first event scenario (E1) can only be evaluated for the VTDL approach because other SPEs do not support any sophisticated failure compensation on an operator level. For this scenario, we have selected the *deploy* functionality as the compensation mechanism, which obtains a new VTDL-file from a predefined location and applies the delta update between the topologies. Here, it is sufficient to reroute the traffic from Portugal to Austria, which results in a low duration between the event and the topology update. The event is detected by the watchdog mechanism of the VISP Runtime, which evaluates the availability and connectivity among the individual operators every ten seconds. The detection of the outtake takes on average 5 seconds, but in the worst case this can take up to 10 seconds, depending on the cycle of the watchdog. These changing detection times result in different compensation durations, which is also indicated by the high standard deviation for this scenario. This scenario also does not require any user interactions since the failure compensation is conducted autonomously by the VISP Runtimes.

The next scenario (E2) describes an active resource configuration within the topology which results in a shorter average duration as for E1. For the VTDL approach, it is sufficient to only reroute the data flow between the sensor and the operator, whereas the baseline approach requires the removal and redeployment of two sub-topologies, i.e., for Spain and Austria, which results in an update duration of almost 54 seconds. That is about 18 times as long as for the VTDL approach. The baseline approach also requires five interactions more than the VTDL approach, because it requires two topology removals and two topology uploads.

The last scenario (E3) requires the removal and update of one VISP Runtime for the baseline approach compared to the update of a single operator. For the new instantiation, all operators are newly deployed which results in a six times higher duration compared to the VTDL approach, where only one operator is removed and the updated one is deployed again.

The evaluation of the VTDL approach against the baseline approach which is used for established SPEs shows that the VTDL-based approach can reduce both the duration for applying changes to a topology as well as the required manual interactions.

VII. Related Work

Up to now, there is no suitable topology description approach, which addresses all features identified in Section III. Nevertheless, there are already several topology description approaches, which address the features at least partially as shown in Table III.

For the discussion of the related work, we only analyze whether the topology description approach supports the desired features. This limitation is specifically relevant for the fault tolerance aspect (F3): Although most SPEs consider fault tolerance mechanisms, up to now, there are hardly any approaches which consider this aspect on a language level. In addition to the identified features, we also consider whether the topology description approach allows for an abstract topology description, i.e., no SPE-specific knowledge is required to define new topologies.

For the VTDL, the most related topology description approach is the SPL [15] because the VTDL uses the same structural concepts. The major differences between the SPL and the VTDL are a lack of several features for SPL, like QoS-related attributes and explicit fault tolerance instructions. The most recent version of the SPL already considers the placement of operators on specific computational resources within one System S instance [15]. Nevertheless, the SPL does not support topologies across multiple System S instances.

The continuous query language (CQL) [21] follows the design principles of SQL. This enables domain experts to design individual operators as well as topologies based on an SQL-like syntax without considering any SPE-specific aspects, like deployment restrictions or resource elasticity. The downside of this abstract approach is that none of the next-generation features are considered within the design of CQL and the focus on the SQL-like syntax makes it hard to integrate them into CQL. While CQL provides an easy to use topology design approach, it is not widely supported by SPEs.

TABLE II: Evaluation Results

	VTDL Approach		Baseline Approach	
	duration (ms)	interactions	duration (ms)	interactions
1. New Topology on one Location	54977.33 (σ = 982.03)	3	54977.33 (σ = 982.03)	3
2. New Topology across four Locations	40953.67 (σ = 431.03)	3	68098.00 (σ = 1701.86)	12
3. Network Disruption (E1)	7532.67 (σ = 1921.05)	0	-	-
4. Resource Reconfiguration (E2)	3301.33 (σ = 281.56)	3	54056.33 (σ = 1554.08)	8
5. Single Operator Update (E3)	5688.67 (σ = 363.53)	3	35623.33 (σ = 1169.39)	4

To resolve this issue, an intermediate language called River [34] has been developed, which allows running CQL-based topologies on System S. This abstraction layer follows a similar approach as the Apache Beam project does and also supports the integration of StreamIt [35], a programmatic topology description language, into System S.

The majority of the SPEs only support a code-based topology description. For our analysis, we consider Apache Storm [3] and Apache Spark Streaming [4] as representatives for established SPEs. Nevertheless their features are similar for other SPEs like Apache Apex[5], Apache Flink [36] and to some extent also Apache Kafka[6]. These SPEs already consider basic QoS-related aspects, like parallelism, and can redistribute computational resources at run time for replicated operators. However, none of these SPEs supports structural changes of topologies at run time (F5). Furthermore, Apache Spark Streaming also allows some basic fault tolerance mechanism.

To abstract the code-based topology representation for Apache Storm, the Flux project proposes a Domain Specific Language (DSL). This DSL allows an abstract description of an Apache Storm topology, which can then be translated into a concrete implementation, similar to Apache Beam or River. Additionally, there are also other topology description approaches like REPARA [37] or SPar [38], but they also do not consider the required features for next-generation SPEs. Based on the comparison shown in Table III, one can see that the features discussed in Section III are only partially supported by current topology description approaches, whereas VTDL considers all of the required features.

The large variety of established (System S [2], Apache Storm [3] or Apache Spark [4]) and novel (Heron [5] or CSA [6]) SPEs results in a large variety of topology description approaches. To address this incompatibility issue, the Apache Beam project has been initiated. This project introduces the abstract Beam Model based on the Dataflow model [30], which is then translated into concrete instructions for different SPEs, like Apache Spark [4] or Apache Flink [36]. However, Apache Beam does not support any of the next-generation features which have been identified in Section III. This is mainly because Apache Beam only provides an abstraction layer for other SPEs and therefore implements the least common denominator of the supported SPEs.

Besides the SPE-specific topology description languages, we also evaluated TOSCA [39] as a foundation for the VTDL because TOSCA already considers different QoS aspects and capabilities for applications. Although these applications follow similar concepts as operators, the main focus of the VTDL is on the structural aspects of the stream processing topology and the technical deployment aspects, the main focus of TOSCA, are out of scope for the VTDL and need to be considered by SPEs.

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

[6]http://docs.confluent.io/current/streams/developer-guide.html

VIII. Conclusion

Within this paper, we have motivated the need for a new topology description approach by discussing the features for next-generation SPEs. Based on these features we have introduced the VTDL, which extends the SPL with the required features to support distributed SPAs as well as fine granular QoS constraints for operators. Besides the abstract notation of the VTDL, we also presented a concrete management mechanism, which is required to use the features of the VTDL within SPEs. This management mechanism has been evaluated based on five scenarios and the evaluation shows that the VTDL approach has a significantly lower update duration for updating topologies compared to traditional approaches. Additionally, the evaluation also shows that the VTDL also enables new possibilities for SPAs like the automatic failure compensation.

For our future work, we plan to evaluate the usability of the VTDL based on a user study conducted with domain experts. This user study aims to assess the performance improvement for designing new topologies based on the VTDL. In addition, we plan to extend the integration to other SPEs by integrating the VTDL into the Apache Beam project to leverage Beam's already existing integration capabilities.

References

[1] A. McAfee, E. Brynjolfsson, T. H. Davenport, D. Patil, and D. Barton, "Big data," *The management revolution. Harvard Business Review*, vol. 90, no. 10, pp. 61–67, 2012.

[2] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo, "SPADE: The System S Declarative Stream Processing Engine," in *International Conference on Management of Data (SIGMOD)*. ACM, 2008, pp. 1123–1134.

[3] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy, "Storm@twitter," in *International Conference on Management of Data (SIGMOD)*. ACM, 2014, pp. 147–156.

[4] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, pp. 10–17, 2010.

[5] M. Fu, S. Mittal, V. Kedigehalli, K. Ramasamy, M. Barry, A. Jorgensen, C. Kellogg, N. Lu, B. Graham, and J. Wu, "Streaming@twitter," *IEEE Data Engineering Bulletin*, vol. 38, no. 4, pp. 15–27, 2015.

[6] Z. Shen, V. Kumaran, M. J. Franklin, S. Krishnamurthy, A. Bhat, M. Kumar, R. Lerche, and K. Macpherson, "CSA: Streaming engine for internet of things," *IEEE Data Engineering Bulletin*, vol. 38, no. 4, pp. 39–50, 2015.

TABLE III: Topology Description Approaches

	VTDL	Apache Beam, Dataflow [30]	SPL [15]	CQL [21]	Flux, Apache Storm [3]	Apache Spark Streaming [4]
Deployment Preferences (F1)	✓		(✓)			
QoS Aspects (F2)	✓				(✓)	(✓)
Fault Tolerance (F3)	✓					(✓)
Semantic Annotation (F4)	✓		✓			
Runtime Modification (F5)	✓				(✓)	(✓)
Data Transfer (F6)	✓					
Abstract Description	✓	(✓)	✓	✓	(✓)	

[7] Z. Zhuang, T. Feng, Y. Pan, H. Ramachandra, and B. Sridharan, "Effective multi-stream joining in apache samza framework," in *International Congress on Big Data*. IEEE, 2016, pp. 267–274.

[8] A. Clemm, M. Chandramouli, and S. Krishnamurthy, "DNA: An SDN framework for distributed network analytics," in *International Symposium on Integrated Network Management (IM)*. IEEE, 2015, pp. 9–17.

[9] A. Botta, W. de Donato, V. Persico, and A. Pescapé, "Integration of Cloud Computing and Internet of Things: A Survey," *Future Generation Computer Systems*, vol. 56, pp. 684–700, 2016.

[10] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer, "Network-aware operator placement for stream-processing systems," in *22nd International Conference on Data Engineering, 2006 (ICDE)*, 2006, pp. 49–49.

[11] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters." *HotCloud*, vol. 12, pp. 10–10, 2012.

[12] L. M. Vaquero and L. Rodero-Merino, "Finding your way in the fog: Towards a comprehensive definition of fog computing," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 5, pp. 27–32.

[13] C. Hochreiner, M. Vögler, S. Schulte, and S. Dustdar, "Elastic Stream Processing for the Internet of Things," in *9th International Conference on Cloud Computing (CLOUD)*. IEEE, 2016, pp. 100–107.

[14] European Comission, "Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC," *Official Journal of the European Union*, vol. 119, pp. 1–88, 2016-05-04.

[15] M. Hirzel, S. Schneider, and B. Gedik, "SPL: An extensible language for distributed stream processing," *ACM Transactions Programming Languages and Systems*, vol. 39, no. 1, pp. 5:1–5:39, Mar. 2017.

[16] S. Schulte, P. Hoenisch, C. Hochreiner, S. Dustdar, M. Klusch, and D. Schuller, "Towards process support for cloud manufacturing," in *18th International Enterprise Distributed Object Computing Conference*, 2014, pp. 142–149.

[17] S. Nakajima, "Introduction to TPM: Total Productive Maintenance," *Productivity Press, Inc.*, 1988.

[18] S. Kolozali, M. Bermúdez-Edo, D. Puschmann, F. Ganz, and P. M. Barnaghi, "A Knowledge-Based Approach for Real-Time IoT Data Stream Annotation and Processing," in *2014 IEEE International Conference on Internet of Things, IEEE Green Computing and Communications, and IEEE Cyber, Physical and Social Computing (iThings/GreenCom/CPSCom 2014)*. IEEE, 2014, pp. 215–222.

[19] Y. Simmhan, B. Cao, M. Giakkoupis, and V. K. Prasanna, "Adaptive Rate Stream Processing for Smart Grid Applications on Clouds," in *2nd International Workshop on Scientific Cloud Computing (ScienceCloud '11)*. ACM, 2011, pp. 33–38.

[20] R. Cortés, X. Bonnaire, O. Marin, and P. Sens, "Stream Processing of Healthcare Sensor Data: Studying User Traces to Identify Challenges from a Big Data Perspective," in *4th International Workshop on Body Area Sensor Networks (BASNet-2015)*, ser. Procedia Computer Science, vol. 52, 2015, pp. 1004–1009.

[21] A. Arasu, S. Babu, and J. Widom, "The CQL continuous query language: semantic foundations and query execution," *Proceedings of the VLDB Endowment*, vol. 15, no. 2, pp. 121–142, 2006.

[22] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. B. Zdonik, "Scalable distributed stream processing," in *First Biennial Conference on Innovative Data Systems Research (CIDR 2003)*. www.cidrdb.org, 2003, pp. 257–268.

[23] M. Villamizar, O. Garces, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano *et al.*, "Infrastructure cost comparison of running web applications in the cloud using AWS lambda and monolithic and microservice architectures," in *16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE, 2016, pp. 179–182.

[24] C. Hochreiner, M. Vögler, P. Waibel, and S. Dustdar, "VISP: An Ecosystem for Elastic Data Stream Processing for the Internet of Things," in *20th International Enterprise Distributed Object Computing Conference*. IEEE, 2016, pp. 19–29.

[25] T. Heinze, L. Aniello, L. Querzoni, and Z. Jerzak, "Cloud-based data stream processing," in *8th International Conference on Distributed Event-Based Systems (DEBS)*. ACM, 2014, pp. 238–245.

[26] A. Rodriguez, R. McGrath, Y. Liu, and J. Myers, "Semantic management of streaming data," in *2nd International Conference on Semantic Sensor Networks*, ser. CEUR Workshop Proceedings, vol. 522. CEUR-WS, 2009, pp. 80–95.

[27] W. Wang, P. Barnaghi, G. Cassar, F. Ganz, and P. Navaratnam, "Semantic sensor service networks," in *IEEE Sensors*. IEEE, 2012, pp. 1–4.

[28] G. Jacques-Silva, B. Gedik, R. Wagle, K.-L. Wu, and V. Kumar, "Building user-defined runtime adaptation routines for stream processing applications," *Proceedings of the VLDB Endowment*, vol. 5, no. 12, pp. 1826–1837, 2012.

[29] F. Baude, L. El Beze, and M. Oliva, "Towards a flexible data stream analytics platform based on the gcm autonomous software component technology," in *International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 2016, pp. 34–41.

[30] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt *et al.*, "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*, vol. 8, no. 12, pp. 1792–1803, 2015.

[31] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Integrating scale out and fault tolerance in stream processing using operator state management," in *International Conference on Management of Data (SIGMOD)*. ACM, 2013, pp. 725–736.

[32] J.-H. Hwang, Y. Xing, U. Cetintemel, and S. Zdonik, "A cooperative, self-configuring high-availability solution for stream processing," in *23rd International Conference on Data Engineering (ICDE)*. IEEE, 2007, pp. 176–185.

[33] F. Keller and S. Wendt, "FMC: An approach towards architecture-centric system development," in *10th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*. IEEE, 2003, pp. 173–182.

[34] R. Soulé, M. Hirzel, B. Gedik, and R. Grimm, "River: an intermediate language for stream processing," *Software: Practice and Experience*, vol. 46, no. 7, pp. 891–929, 2016.

[35] W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A language for streaming applications," in *International Conference on Compiler Construction*. Springer, 2002, pp. 179–196.

[36] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache Flink™: Stream and Batch Processing in a Single Engine," *Data Engineering Bulletin*, vol. 38, no. 4, pp. 28–38, 2015.

[37] M. Danelutto, T. De Matteis, G. Mencagli, and M. Torquati, "Data stream processing via code annotations," *The Journal of Supercomputing*, pp. 1–15, 2016.

[38] D. Griebler, M. Danelutto, M. Torquati, and L. G. Fernandes, "SPar: A DSL for High-Level and Productive Stream Parallelism," *Parallel Processing Letters*, vol. 27, no. 1, 2017.

[39] T. Binz, U. Breitenbücher, O. Kopp, and F. Leymann, "TOSCA: Portable Automated Deployment and Management of Cloud Applications," in *Advanced Web Services*. Springer, 2014, ch. 22, pp. 527–549.