Evaluating a Flow-Based Programming Approach as an Alternative for Developing CEP Applications in IoT

Guadalupe Ortiz, Iván Castillo, Alfonso Garcia-de-Prado, and Juan Boubeta-Puig

***Abstract*—One of the main advantages brought by the Internet of Things (IoT) is the possibility of having large amounts of data from several sources that allow us, once analyzed, to make decisions in various domains in real time. This implies the need to be able to process large volumes of data in more or less limited processing times depending on the application domain. In this sense, complex event processing (CEP), used in conjunction with an enterprise service bus (ESB), has proven to be very efficient in multiple domains. In search for greater efficiency, some CEP engines offer the option of using flow-based programming (FBP) rather than their traditional programming using CEP together with an event bus. However, its use, while it may be more efficient, can lead to other limitations. In this article, we analyze and describe the performance and limitations of using a CEP engine with an ESB versus a CEP engine with FBP. This will allow developers to decide which option is more convenient for their IoT system depending on the application domain and its specific needs.**

***Index Terms*—Complex event processing (CEP), dataflow, dataflow computing, enterprise service bus (ESB), flow-based programming (FBP), Internet of Things (IoT).**

I. Introduction

CURRENTLY, large amounts of data are generated from multiple sources that are waiting to be processed in order to obtain greater knowledge of the domain in question and to make profitable decisions. Not surprisingly, many of the current research works tackle how to develop data-centric architectures for data processing in general and for the processing of data from the Internet of Things (IoT) in particular [1]. It is already unthinkable to focus on batch processing, yet the processing of data from the IoT is expected to be done in streaming to facilitate real-time decision making.

Multiple publications endorse complex event processing (CEP) as a successful technology for streaming data processing at the IoT [2]–[5], including a wide variety of work, from those focused on IoT data traffic analysis [6], [7] to those that even go as far as to provide a visual language for programming CEP patterns for IoT domains [8], [9]. Indeed, as stated by Rahmani *et al.* [10], CEP has become a key part of the IoT. This integration of CEP with the IoT is not only conducted in the cloud, but also at levels closer to the device, such as fog or edge [11]. Some of these proposals benefit from the integration of CEP with an enterprise service bus (ESB). An ESB eases the resolution of conflicts between hardware (e.g., IoT devices) and software (e.g., the CEP engine) [12] and plays a key role in multiple application domains in general [13] and in the IoT [14] and Industrial IoT (IIoT) [15] ones in particular. Although the performance results are satisfactory, we cannot lose sight of the fact that we can find scenarios which call for greater performance, as well as the inevitable growth in the amount and speed of data generated due to the improvement of cyber–physical devices as well as communications. To respond to this need, some CEP languages offer an alternative to the *traditional programming* of CEP-based systems, which require an event bus [16], through the support of dataflow programming and flow-based programming (FBP).

Dataflow programming is a programming paradigm that models a program as a directed graph of the data flowing between operations supported by FBP. FBP approaches an application not as a single, sequential process, but as a set of asynchronous processes that communicate through flows. This allows the developer to be able to focus on the application data and the transformations applied to them to produce the expected results.

Since FBP processes continue to run as long as they have data to work on, FBP applications are generally more efficient than other conventional programs, and they optimize the use of the machine's processor. If so, using dataflow-based CEP can provide adequate performance in domains where the traditional use of CEP integrated with an event bus, such as an ESB, can lead to latency problems [14]. However, the complexity of FBP may impose limitations that are not encountered when using CEP with an ESB. Thus, depending on the system, we will have to consider whether to use a CEP engine with

Manuscript received August 11, 2021; revised September 27, 2021; accepted November 19, 2021. Date of publication November 24, 2021; date of current version June 23, 2022. This work was supported in part by the Spanish Ministry of Science and Innovation and the European Regional Development Fund (ERDF) through Project FAME under Grant RTI2018-093608-B-C33; in part by the Andalusian Local Government and ERDF funds through Project DECISION under Grant P20_00865; in part by the Excellence Network RCIS under Grant RED2018-102654-T; and in part by the Research Plan from the University of Cadiz and Grupo Energético de Puerto Real S.A. through Project GANGES under Grant IRTP03_UCA. *(Corresponding author: Guadalupe Ortiz.)*

Guadalupe Ortiz, Iván Castillo, and Juan Boubeta-Puig are with the Department of Computer Science and Engineering, University of Cadiz, 11519 Cádiz, Spain (e-mail: guadalupe.ortiz@uca.es; ivan.castillo@alum.uca.es; juan.boubeta@uca.es).

Alfonso Garcia-de-Prado is with the Computer Architecture and Technology Department, University of Cadiz, 11519 Cádiz, Spain (e-mail: alfonso.garciadeprado@uca.es).

Digital Object Identifier 10.1109/JIOT.2021.3130498

ESB despite possible latency losses or a CEP engine supporting FBP despite the possible limitations we may encounter in terms of programming, such as less flexible syntax. This brings us directly to the following research questions (RQ).

- RQ1: Assuming that dataflow-based CEP can be used to implement a system, which permits the detection of situations of interest through real-time processing in streaming; do we find any limitation that was not observed when using CEP with an ESB?
- RQ2: If we can implement the system in RQ1 without major limitations, can we improve the performance of a CEP application integrated with an ESB by using FBP, thus avoiding having to use the ESB; and, if so, how much does it improve our system's performance?
- RQ3: Finally, what would be the advantages and disadvantages of one system over the other, and when would it be more convenient to use either of them?

In order to answer these RQ, the main aims of this article are, on the one hand, to implement two equivalent systems where one uses a CEP with an ESB and the other a CEP with dataflows and FBP, as well as to define a benchmark that allows us to evaluate their performance. Second, to conduct performance tests for both systems using the defined benchmark in the same test environment. Finally, we will analyze the difficulties and limitations in the implementation of both systems and their performance according to the benchmark in question to evaluate in which scenarios it is more convenient to use either. For this purpose, we have chosen to use the Esper CEP engine [17]: Esper is a highly scalable and open-source Java-based software engine for CEP that can rapidly process and analyze large volumes of incoming IoT data in real time. Esper provides the Esper event processing language (EPL), which extends the SQL standard and enables the precise definition of complex event patterns to be detected.

The remainder of this article is organized as follows. Section II presents the background on the paradigms and architectures used in the proposed implementations. Then, Section III explains how both CEP-based implementations have been carried out and what the limitations found were. Afterward, Section IV describes the benchmark defined for their evaluation and Section V explains the results obtained in it. The related work is then analyzed in Section VI. Finally, discussion and conclusions are presented in Section VII.

II. Background

This section introduces CEP and the Esper CEP engine, afterward event-driven service-oriented architecture (SOA) (ED-SOA or SOA 2.0) is explained, and finally, FBP is introduced.

A. Complex Event Processing

CEP [18] is a powerful technology that allows us to capture, analyze, and correlate huge amounts of heterogeneous data (in the form of *simple events*) in order to promptly detect relevant situations in a particular domain [19]. To this end, this technology requires the definition of a set of event patterns specifying the conditions to be met from the content of the events of one or more incoming data streams. The situation of interest detected by an event pattern is named *complex event*. It is referred to as a complex event because it is obtained from the analysis and correlation of one or several simple events in a given period of time. A CEP engine is the software responsible for the real-time analysis of streaming data according to the defined patterns. The Esper CEP engine stands out because of its performance and maturity, and the wide coverage of its supported EPL for event pattern definition [20]. Esper EPL is a declarative and data-oriented language, which is compliant to the SQL-92 standard and extended for analyzing event streams and with regards to time. Since Esper compiles EPL source code into Java virtual machine (JVM) bytecode, the code can run on a JVM within the Esper runtime environment.

Esper can be operated under two different programming paradigms. On the one hand, to work with event streams in Esper, we have traditionally needed an infrastructure that supports reading the data streams from the sources, transforming them into the appropriate formats and managing the complex events detected by the CEP engine to deliver them to the target user. Usually, this work has been done by an ESB, as part of a SOA 2.0. On the other hand, Esper supports the programming of dataflows in EPL and, therefore, supports FBP. It provides aid to integrate input and output adapters in runtime. These adapters can be those provided by Esper IO or developed *ad hoc* by the programmer. By making use of the dataflows, we eliminate the need to use an external ESB; as a result, an improvement in system performance is expected. Both paradigms—SOA 2.0 and FBP—are explained in the following sections.

B. Event-Driven Service-Oriented Architectures

SOA provides a paradigm for the design and implementation of loosely coupled distributed systems. In such an architecture, services are the major implementation mechanism. Such services should provide a well-defined interface that delivers communications based on a standard protocol. The use of these architectures facilitates and makes the interoperability among third-party systems in a loosely coupled way more flexible. This way, the developer can maintain the focus on the business process, rather than on the selected technologies for implementation. As a consequence, maintenance cost is minimized and the system can evolve more easily [21].

The SOA 2.0 term was originally introduced in 2006 [22], [23], to refer to a combination of a SOA with an event-driven architecture (EDA). While in traditional SOAs communications are mainly done through remote procedure calls, communication between users, applications, and services in a SOA 2.0 is usually done through events [18], and it is in this type of architectures where CEP plays a relevant role, facilitating the detection of situations of interest in SOA 2.0 and the processing of IoT data in such an architecture.

For a successful integration of service-based systems, an infrastructure that permits a flexible interconnection among application and messaging middlewares is required [13], [21]. These requirements are fulfilled by an ESB. The ESB provides other additional advantages, including guaranteed system scalability, and allows the integration of several heterogeneous

data sources and invocations to various distributed systems and applications [24]. It also facilitates the connection of the input data streams with the CEP engine and the output of the latter with the output data streams. All these features, especially the ease of connecting data streams and invocations from various sources and systems, are what have made the ESB a strong candidate for an IoT architecture [12]–[15].

C. Flow-Based Programming

FBP is a programming paradigm that was first introduced in the 1970s by J. Paul Rodker Morrison. However, it is currently gaining in popularity since dataflow processing is a key requirement for cutting-edge data-driven applications [25]. A data-driven application can be created as a network of asynchronous processes that exchange data chunks and apply transformations to them.

This paradigm models software systems as a directed graph of processing nodes [26]. These processes are executed asynchronously and communicate with each other through message passing. Processes are executed in several processing nodes, which are interconnected. Each processing node is responsible for conducting part of the main computation of the application while, on the other hand, dataflow dependencies between nodes have to be defined. When a node receives data, then the computation is triggered.

Modularity is one of the main advantages of FBP, allowing a system's components to be combined, separated, and reused by changing their interconnections. Other benefits of this paradigm are code reuse, exceptional composability, implicit pipeline parallelism, and testability [25]. Since CEP is normally used for processing streaming data flows, the synergy between FBP and CEP is evident, and also how a CEP engine integrates an FBP programming model can benefit from the former advantages.

III. System's Implementations and Limitations

In this section, we explain two equivalent implementations for CEP with Esper 8.4.0, based on the integration of CEP in an ESB, on the one hand, and on the use of CEP through FBP on the other. Afterward, the limitations found during the implementation are explained.

A. Implementation Integrating CEP in an ESB

We have prepared the implementation of a system that integrates CEP with an ESB, as shown in Fig. 1. In particular, we have integrated Esper CEP with Mule ESB, as successfully done in the past [4], [5]. Note that by using an ESB, all modules necessary to receive and process data from multiple sources and to notify interested parties of detected situations of interest can be integrated into it; that is, the CEP engine and the source for the input data and the sink for the output data, which are connected to the respective brokers—which may be located on another machine. In general, the flow of the system consists of a series of data arriving at the ESB from a message broker in a format supported by the CEP engine and the complex events detected in the latter are sent to an outgoing message broker. Although a single broker could be used for data input and output, two different brokers were used

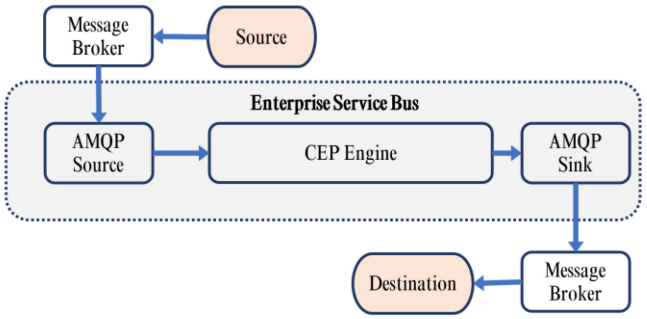

Fig. 1. Representative flow of an implementation integrating CEP with ESB.

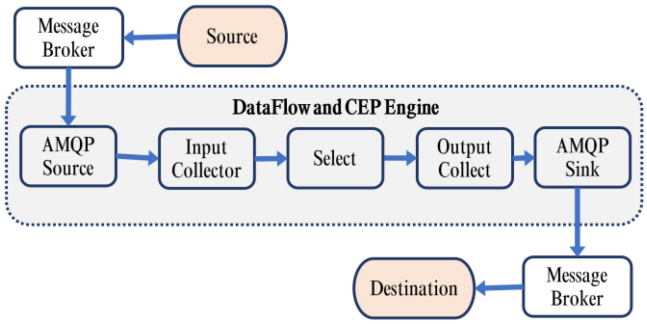

Fig. 2. Representative flow of an implementation using flow-based CEP.

for the tests to avoid the broker overload influencing the final system performance. According to the common technologies chosen for the implementation of both systems, the process would be as follows.

1) On the one hand, the input data stream will reach the ESB from a RabbitMQ messaging broker using the AMQP 0.9 protocol through an AMQP source in the ESB.
2) Since the received messages are in a JSON format supported by the CEP engine, we do not require any format transformation, so from the AMQP source they are directly submitted to the Esper CEP engine.
3) Afterward, the CEP engine will process such events and, when a complex event is detected, it will send all the information associated to the latter to an AMQP Sink in the ESB linked to a RabbitMQ message broker, again through the AMQP 0.9 protocol.

Please note that the messages have been instrumented to obtain the processing capacity, CPU usage, and RAM memory consumed during the execution.

B. Implementation Integrating CEP With Dataflows

Second, we have implemented the system using the FBP paradigm with CEP, as shown in Fig. 2. Again, we will use a different broker for data input and output when evaluating performance. The implemented flow in Esper CEP contains the source for the input data and the sink for the output data, which are connected to the external brokers, as well as the collectors required by the CEP engine for using dataflows, as explained as follows.

1) It will have an AMQP source, which reads the incoming message from the message queue in the RabbitMQ broker.

2) The events will be read by a collector—*AMQPToObjectCollector*—which permits transforming the received events into the event objects supported by the CEP engine.
3) Then, the output of the mentioned collector will be directed to a *select* operator, which will permit the application of a pattern to the incoming events and submission of the detected complex events to an output collector.
4) The complex events detected in the previous step are submitted to an output collector—*ObjectToAMQPCollector*—which will convert them into the right format to be submitted to an AMQP Sink.
5) Finally, the outgoing events of the last collector are submitted to an AMQP Sink, which collects the complex events detected and resubmits them to the Rabbit MQ output message broker, to direct them to the final destination.

Note that the collector will also allow us to instrument the messages to obtain the processing capacity, CPU usage, and RAM memory consumed during the execution.

C. Limitations of the Implementations

For simplicity, from now on we will call the implementation described in Section III-A CEP-ESB and the one described in Section III-B will be referred to as CEP-Dataflow. Even though we have implemented equivalent systems with both types of architectures, we have to mention we found some limitations in the CEP-Dataflow one.

First, unlike CEP-ESB architectures, with Esper dataflows we had to define the complete flow of the program in a single EPL sentence, which makes it difficult to connect one pattern's output to the input of another.

Second, EPL dataflow patterns are *black boxes*, and they do not share any insight among patterns, which hinder the possibility of personalizing the event types at runtime. However, with the CEP-ESB implementation, it is possible to add new event types and patterns easily at runtime.

Third, we were able to work directly with the events received in JSON format in the CEP-ESB implementation, whereas the CEP-Dataflow required a transformation to another format; specifically, we chose Java Map.

Fourth, the routing of messages is more complicated in CEP-Dataflow; not surprisingly, since this is one of the basic functions of an ESB. With CEP-Dataflow, the use of an internal event bus with additional configurations and requirements is necessary.

Fifth, in CEP-Dataflow, the definition of an event type at runtime is only valid in the scope of a specific flow; however, in CEP-ESB, the definition of an event type is by default valid for any pattern of any flow in the CEP engine.

Finally, while CEP-ESB offers many predefined input/output operators for the CEP engine, CEP-Dataflow offers very little variety, which implies the need for the developer to implement new connectors in case he/she wants to connect the input or output using a different protocol than those provided by Esper, which is not a trivial task.

On the other hand, the CEP-ESB implementation is much more flexible when programming the application, overcoming, as we have seen, the limitations of the CEP-Dataflow implementation. However, CEP-ESB presents a greater limitation in terms of system resource consumption: ESBs are characterized by making system integration much easier at the cost of higher resource consumption. As an example, Mule ESB requires at least 1-GB RAM and 4-GB storage [27], compared to the few MB of RAM and 500-MB storage required by Esper [28]. This involves that in certain IoT scenarios where the system is to be deployed on an edge device, the choice of the CEP-ESB option may compromise the performance of the system by saturating the system's resources. Naturally, the consumption of both applications in execution will depend on the number of streams and the type of statements deployed. Moreover, the use of the ESB requires an additional knowledge of system integration for the inclusion of the CEP engine in the ESB.

IV. Evaluation

Once we have the two implementations, we are going to evaluate their performance. In this section, we first describe the resources used for the tests as well as the procedure followed; then, we explain the patterns defined for the benchmark.

A. Resources and Procedure

We have carried out the tests with two sets of resources. In *Configuration 1*, we have used the following machines.

1) A PC with an Intel i5-4570T processor and 8 GB of RAM, in which we installed RabbitMQ 3.6.10. This will be the input queue for the system.
2) A PC with an i7-3770 CPU and 8 GB of RAM, which acts as our main system server. We used it both for the CEP-ESB and CEP-Dataflow implementation, of course one at a time.
3) A server machine with an Intel Xeon Silver 4110 processor and 32 GB of RAM. This one was used as output queue.
4) One more PC with an Intel i3 3220T processor and 4 GB of RAM was used to submit the test data to the inbound queue.

Assuming that in a specific IoT scenario, we need to deploy a small and cost-effective device, we have used a Raspberry Pi 3 Model B Rev 1.2 for data processing in *Configuration 2* in both the CEP-ESB and CEP-Dataflow implementations. The Raspberry Pi processor is a Broadcom BCM2837B0, Cortex-A53 (ARMv8) 64-bit SoC at 1.4 GHz, with 1 GB of LPDDR2 SDRAM and Ethernet connection over USB 2.0 at 300 Mb/s. The rest of the machines (input and output queue ones and the one that submits the test data to the inbound queue) remain the same as in the initial configuration.

The tests will consist of deploying a CEP pattern in the implementation under evaluation, submitting events to the system with different incoming ratios to check for any variations in the system's behavior when stressed, and measuring the CPU and RAM memory usage as well as the system's capacity to process all the incoming events. Concerning the

```
create schema BenchmarkEvent
  (attr1 string, attr2 string, attr3 string,
    proccesedTimestamp long, receivedTimestamp long)
```

Listing 1. Benchmark type of events.

```
@Name('Statement1')
INSERT INTO Statement1
  SELECT *
  FROM BenchmarkEvent
```

Listing 2. Statement 1 for the CEP-ESB implementation.

procedure followed to carry out the tests, it is important to mention that:

1) each test will last 10 min, regardless of the timing of the messages;
2) between tests, we have ensured that we start from a clean instance of the system, that is, that the system is not affected by underlying tests;
3) once we start the system, some messages are submitted for 2–3 min as warm-up, before setting the initial test time;
4) after each test is completed, the results are collected;
5) each test was repeated three times; afterward, the average of the three executions was calculated.

B. Benchmark Patterns

To test a CEP environment, it is necessary to define, on the one hand, the type of events that the system is going to receive (in Esper, this is done by defining its scheme) and, on the other hand, the patterns that are going to be applied to the stream of incoming events to see if they are fulfilled.

In particular, we have defined the type *BenchmarkEvent*, as follows.

As we can see in Listing 1, the BenchmarkEvent type is composed of the following fields.

1) *attr1*, *attr2*, *attr3:* These are the event attributes that will permit us to test the functionality of the different patterns defined for the tests.
2) *Receivedtimestamp:* This is a timestamp used to record the time when the simple event reaches the CEP engine.
3) *Processedtimestamp:* This timestamp is used to record the time when the CEP engine has finished processing the event.

The pattern selected for the benchmark will make use of different Esper operators that can give rise to diverse computer system loads. Note that Esper provides a very large number of operators, of which we have selected several, of frequent use and diverse computational load. In particular, six patterns have been defined, which increase their computational complexity progressively. The patterns have been written in an analogous way both for CEP-ESB and CEP-Dataflow implementations.

1) Statement 1: The first pattern we have tested selects all messages without any kind of discrimination, which means that we will have an output complex event for each input simple event immediately after its reception. We can see both implementations in Listings 2 and 3.

```
Select(demostream) ->
  outdemostream
  { select: (select * from demostream )  }
```

Listing 3. Statement 1 for the CEP-Dataflow implementation.

```
@Name('Statement2')
INSERT INTO Statement2
  SELECT *
  FROM BenchmarkEvent BEv1
  WHERE BEv1.attr1 = "Attribute"
```

Listing 4. Statement 2 for the CEP-ESB implementation.

```
Select(demostream) ->
  outdemostream
  { select: (select * from demostream
    where attr1 = "Attribute")  }
```

Listing 5. Statement 2 for the CEP-Dataflow implementation.

```
INSERT INTO Statement3
  SELECT *
  FROM BenchmarkEvent.win:length(1) BEv1
  WHERE BEv1.attr1 = "Attribute"
```

Listing 6. Statement 3 for the CEP-ESB implementation.

```
Select(demostream) ->
  outdemostream
  { select: (select * from demostream.win:length(1)
    where attr1 = "Attribute")  }
```

Listing 7. Statement 3 for the CEP-Dataflow implementation.

```
@Name('Statement4')
INSERT INTO Statement4
  SELECT *
  FROM BenchmarkEvent.win:time(2 min) BEv1
  WHERE BEv1.attr1 = "Attribute"
```

Listing 8. Statement 4 for the CEP-ESB implementation.

2) Statement 2: The second pattern we have tested selects only the incoming messages whose *attr1* value is *Attribute*; therefore, we will have an output complex event for each input simple event with *attr1* valued *Attribute* immediately after its reception. We can see both implementations in Listings 4 and 5.

3) Statement 3: In this third pattern, we have added a length sliding window to Statement 2; that is, the window keeps the last events occurred according to the size of the window in memory. In this case, the window length is 1 event and therefore, we will have an output complex event for each simple input event with *attr1* valued *Attribute* after a window of length 1 is closed. We can see both implementations in Listings 6 and 7.

4) Statement 4: In this fourth pattern, we have added a sliding temporal window to Statement 2, that is, the window keeps the events occurred during the time specified in the window in memory. In this case, we will have a complex event output for each simple input event with *attr1* valued *Attribute* produced in the last two minutes. We can see both implementations in Listings 8 and 9.

```
Select(demostream) ->
  Outdemostream
  { select: (select * from demostream.win:time(2 min)
    where attr1 = "Attribute") }
```

Listing 9. Statement 4 for the CEP-Dataflow implementation.

```
@Name('Statement5')
INSERT INTO Statement5
  SELECT BEv1.attr1 as attr1,
       COUNT (DISTINCT BEv1.attr2) as eventTotal,
       BEv1 receivedTimestamp as receivedTimestamp
  FROM BenchmarkEvent.win:time(2 min) BEv1
  WHERE BEv1.attr3 = "Attribute"
```

Listing 10. Statement 5 for the CEP-ESB implementation.

```
Select(demostream) ->
  outdemostream
  { select: (select attr1, count(distinct attr2) as
eventTotal, receivedTimestamp
    from demostream.win:time(2 min)
    where attr3 = "Attribute") }
```

Listing 11. Statement 5 for the CEP-Dataflow implementation.

```
@Name('Statement6')
INSERT INTO Statement6
  SELECT BEv1.attr1 as attr1,
    COUNT (DISTINCT BEv1.attr2) as eventTotal,
    BEv1.receivedTimestamp as receivedTimestamp
  FROM BenchmarkEvent.win:time(2 min) BEv1
  WHERE BEv1.attr3 = "Attribute"
  GROUP BY BEv1.attr1;
```

Listing 12. Statement 6 for the CEP-ESB implementation.

```
Select(demostream) ->
  outdemostream
{ select: ( select attr1, count(distinct attr2) as
eventTotal, receivedTimestamp
    from demostream.win:time(2 min)
    where attr3 = "Attribute" group by attr1 )}
```

Listing 13. Statement 6 for the CEP-Dataflow implementation.

5) Statement 5: In order to further increase the computational complexity, in this pattern, we have added the sentence *COUNT DISTINCT* to Statement 4. This way, we will have a complex event output for the simple input events with *attr3* valued *Attribute* produced in the last 2 min. The output will be composed of complex events with three attributes: *attr1* and *eventTotal*—calculated as the number of simple events whose *attr2* have distinct values—and *receivedTimestamp*. We can see both implementations in Listings 10 and 11.

6) Statement 6: Finally, we have added the *GROUP BY* clause to Statement 5. This clause divides the output of a pattern into groups. This way, we will have a similar complex event output as in Statement 5, grouped by the *attr1* property. We can see both implementations in Listings 12 and 13.

V. Evaluation Results

This section shows the results obtained in the performance tests with Configurations 1 and 2 with each of the patterns explained in Section IV.

TABLE I
Performance of Statement 1 for CEP-ESB and CEP-Dataflow Implementations

Incoming Rate (events/s)	CEP-ESB Implementation			CEP-Dataflow Implementation		
	Events Processed (%)	Memory Usage (MB)	CPU Usage (%)	Events Processed (%)	Memory Usage (MB)	CPU Usage (%)
100	100	1331.47	2.30	100	385.81	0.44
1 000	100	1337.04	21.17	100	722.85	0.46
2 500	100	1341.35	36.99	100	650.06	0.75
5 000	100	1342.91	54.13	100	646.38	2.89
10 000	90.9	1346.28	87.36	100	711.79	5.66
15 000	-	-	-	78.1	875.51	6.53

TABLE II
Performance of Statement 2 for CEP-ESB and CEP-Dataflow Implementations

Incoming Rate (events/s)	CEP-ESB Implementation			CEP-Dataflow Implementation		
	Events Processed (%)	Memory Usage (MB)	CPU Usage (%)	Events Processed (%)	Memory Usage (MB)	CPU Usage (%)
100	100	1342.81	3.68	100	377.80	0.23
1 000	100	1336.26	24.82	100	723.50	1.32
2 500	100	1337.49	38.83	100	500.85	2.43
5 000	100	1343.88	54.25	100	663.05	3.86
10 000	93.72	1343.33	92.83	100	752.78	8.23
15 000	-	-	-	97.87	947.19	7.1

A. Results Obtained With Configuration 1

As previously explained, the patterns in Listings 2 and 3 generate a complex event from each simple event entering the system. In Table I, we can find the results of the tests performed with this pattern for the CEP-ESB implementation, as for the CEP-Dataflow one. After performing the tests, we could draw the conclusion that a similar performance is obtained with both technologies; however, CPU consumption is much lower in the CEP-Dataflow implementation (5.66% compared to 87.36% of use). The same happens with RAM consumption, which is lower in the CEP-DataFlow implementation. Furthermore, the CEP-ESB implementation does not support real-time processing of an incoming rate of 10 000 events per second. We find the CEP-Dataflow limits at 15 000 events per second, but CPU and RAM consumption remain low; it is not the CEP engine that is limiting the input rate but the network on which this architecture runs.

A *WHERE* clause has been added in the patterns in Listings 4 and 5. Such a clause will select the simple events whose attribute value is *Attribute*. It should be noted that we have evaluated whether the fact that the condition is met or not impacts on the CEP engine's performance and no variation has been noticed; that is, performance is the same whether 50% or 100% of the messages meet the condition. Table II shows the results for the test.

As was the case with the first test, both programming approaches show similar performances regarding the percentage of processed events. However, Esper Dataflows only requires 8.23% of CPU usage for the computation of events, compared to 92.83% of the implementation with Mule ESB with an incoming rate of ten events per second.

To increase the computational complexity of CEP, we have started to add windows from this test on. For the patterns

TABLE III
PERFORMANCE OF STATEMENT 3 FOR CEP-ESB AND CEP-DATAFLOW IMPLEMENTATIONS

Incoming Rate (events/s)	CEP-ESB Implementation			CEP-Dataflow Implementation		
	Events Processed (%)	Memory Usage (MB)	CPU Usage (%)	Events Processed (%)	Memory Usage (MB)	CPU Usage (%)
100	100	1327.52	2.91	100	389.81	0.34
1 000	100	1336.16	36.84	100	723.63	0.83
2 500	100	1339.43	49.26	100	725.99	0.76
5 000	100	1339.78	88.83	100	743.11	5.63
10 000	52.08	1339.29	94.77	100	1066.50	6.80
15 000	-	-	-	90.71	896.94	6.36

TABLE IV
PERFORMANCE OF STATEMENT 4 FOR CEP-ESB AND CEP-DATAFLOW IMPLEMENTATIONS

Incoming Rate (events/s)	CEP-ESB Implementation			CEP-Dataflow Implementation		
	Events Processed (%)	Memory Usage (MB)	CPU Usage (%)	Events Processed (%)	Memory Usage (MB)	CPU Usage (%)
100	100	1354.23	3.28	100	400.01	0.33
1 000	100	1359.65	23.53	100	768.97	0.65
2 500	100	1352.47	52.37	100	976.89	1.08
5 000	100	1360.64	90.22	100	1228.8	4.41
10 000	50.80	1359.14	94.74	100	1841.53	9.21
15 000	-	-	-	75.46	1966.37	54.7

TABLE V
PERFORMANCE OF STATEMENT 5 FOR CEP-ESB AND CEP-DATAFLOW IMPLEMENTATIONS

Incoming Rate (events/s)	CEP-ESB Implementation			CEP-Dataflow Implementation		
	Events Processed (%)	Memory Usage (MB)	CPU Usage (%)	Events Processed (%)	Memory Usage (MB)	CPU Usage (%)
100	100	1335.37	3.95	100	413.07	0.29
1 000	100	1338.69	24.46	100	814.52	1.54
2 500	100	1362.15	58.37	100	1066.03	4.52
5 000	99.84	1356.17	94.17	100	1287.33	5.26
10 000	47.73	1352.99	93.24	100	1831.63	8.55
15 000	-	-	-	57.60	1878.46	22.71

TABLE VI
PERFORMANCE OF STATEMENT 6 FOR CEP-ESB AND CEP-DATAFLOW IMPLEMENTATIONS

Incoming Rate (events/s)	CEP-ESB Implementation			CEP-Dataflow Implementation		
	Events Processed (%)	Memory Usage (MB)	CPU Usage (%)	Events Processed (%)	Memory Usage (MB)	CPU Usage (%)
100	100	1326.08	3.28	100	387.86	0.35
1 000	100	1322.60	21.33	100	777.49	0.51
2 500	100	1324.72	55.37	100	1086.04	2.39
5 000	100	1320.08	93.46	100	1306.92	4.72
10 000	51.37	1324.95	94.58	100	1840.68	9.58
15 000	-	-	-	73.33	1904.02	39.55

in Listings 6 and 7, we keep the *WHERE* clause and add a length window. Table III shows the results of the various tests that were performed for both paradigms. We can notice the performance impact of adding windows to the CEP-ESB implementation, since the maximum rate supported by the system is limited to 5208 events per second. Furthermore, the increase in CPU consumption with respect to the results of the analogous tests in Tables I and II can be highlighted. On the other hand, RAM consumption remains rather stable compared to previous tests.

Nevertheless, CPU consumption in the CEP-DataFlow implementation is lower (6.36% maximum) compared to the previous test's maximum values (Table II), although there is a peak at 10 000 events per second due to background tasks. We must take into account that even though we have tried to minimize the impact, CPU metrics may show slight variations when having several system processes in the background. However, no increase in the main memory usage is noticed.

In the next test, a time window is used instead of a length one, as shown in the patterns in Listings 8 and 9. The results of the test are shown in Table IV. In this case, memory consumption has increased in the CEP-DataFlow implementation, which is due to the fact that when time windows are used, the events have to be stored in memory for a period of time determined in the pattern. However, performance is again better in the CEP-Dataflow implementation than in the CEP-ESB one (10 000 versus 5080 events per second, respectively). In addition, we can highlight that even increasing the message rate in the CEP-Dataflow implementation, memory consumption is ten times lower compared to the implementation with CEP-ESB. At a rate of 15 000 events per second, CEP-Dataflow increases consumption to 54.7% of CPU and 1966.37 MB of memory. This drastic increase in CPU and memory usage is due to the fact that as we increase the rate of input events per second, more data have to be stored in each time window, and therefore, there is greater memory consumption and processor context switches take more time. There is also a greater transfer of input/output data and, for all of the above, a greater saturation of the CPU that begins to have unstable behavior due to the high workload.

In order to further increase the complexity, a *COUNT DISTINCT* clause has been added to the previously tested patterns, as shown in the patterns in Listings 10 and 11. The results of the evaluation are shown in Table V. We can check that system response is similar to that of the previous test (see Table IV): we have once more lower memory and CPU consumption in the CEP-Dataflows implementation as well as a higher capacity to support high incoming rates. At 15 000 events per second of input rate, the CPU consumption in the CEP-Dataflow implementation is contained (22.71%) and the memory consumption reaches 1878.46 MB.

Finally, we added the *GROUP BY* clause to the previous pattern in Listings 12 and 13. In Table VI, we can see that we have obtained similar results to the previous tests, in which the CEP-ESB implementation supported a maximum rate of around 5000 incoming events per second, compared to the 10 000 supported by the CEP-Dataflow implementation. It can be noted that the CPU usage of the CEP-Dataflow implementation remains low, but memory use increases, although not as much as in the CEP-ESB implementation. At a rate of 15 000 events per second, the CPU and memory consumption in the CEP-Dataflow implementation reach 39.55% and 1904.22 MB, respectively.

To facilitate a thorough overview of the evaluation results, Fig. 3 shows the complete comparison for all the statements (St1 to St6) and incoming rates in terms of CPU and memory usage. Fig. 3(a) and (b) clearly shows that memory usage

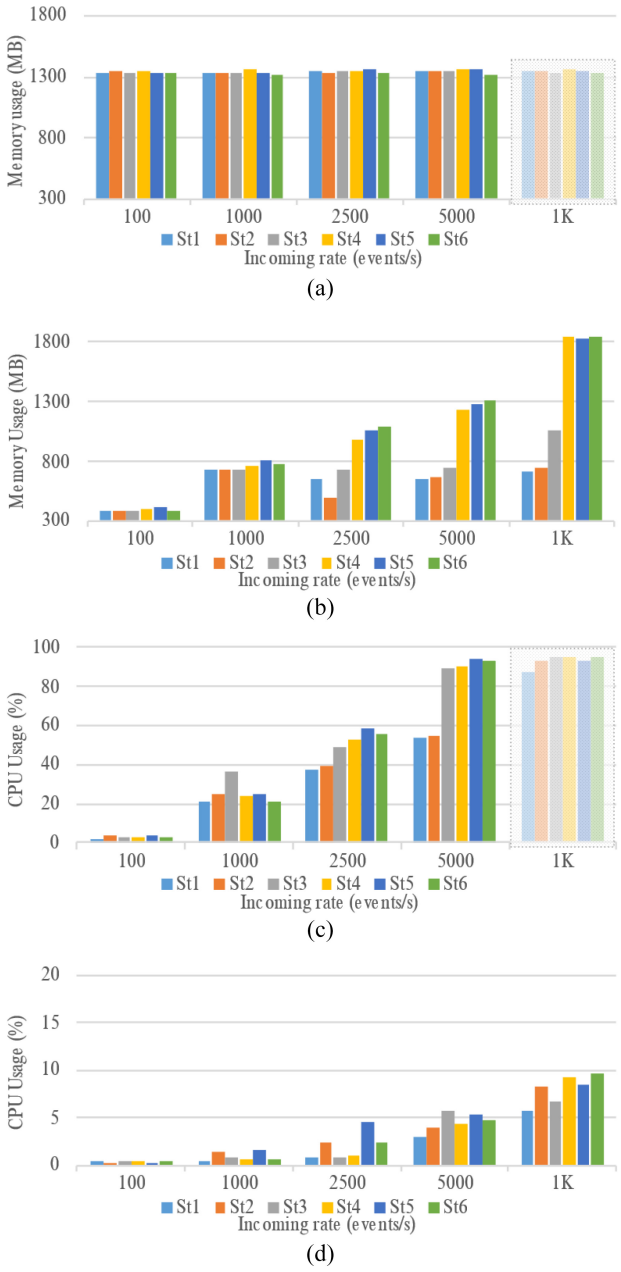

Fig. 3. Performance comparison for all the statements and incoming rates. (a) CEP-ESB memory Usage. (b) CEP-dataflow memory usage. (c) CEP-ESB CPU usage. (d) CEP-dataflow CPU usage.

remains stable, though high, for the CEP-ESB implementation, compared to the CEP-Dataflow implementation, which starts with very low values and increases over time. Recall that CEP-ESB does not support the higher incoming rates for various statements, which CEP-Dataflow does, although with higher memory consumption. Fig. 3(c) and (d) also clearly illustrates that CPU consumption in the CEP-Dataflow implementation is much lower than in the CEP-ESB one. It is worth mentioning that despite testing with patterns that could lead to higher memory consumption, no significant changes have been observed in this regard.

B. Results Obtained With Configuration 2

In this second configuration, it is evident that by processing the data with a much less powerful machine, the ratio of successfully processed input events is lower. Furthermore, the enormous difference in processing capacity between both architectures is clearly visible.

TABLE VII
PERFORMANCE OF ALL THE STATEMENTS FOR CEP-ESB AND CEP-DATAFLOW IMPLEMENTATIONS IN THE RASPBERRY PI

CEP-ESB Implementation				CEP-Dataflow Implementation			
Incoming Rate (events/s)	**Events Processed (%)**	**Memory Usage (MB)**	**CPU Usage (%)**	**Incoming Rate (events/s)**	**Events Processed (%)**	**Memory Usage (MB)**	**CPU Usage (%)**
Pattern 1							
100	100	606.4	29.6	**100**	100	57.7	19.6
200	100	605.7	76.2	**1 000**	100	57.7	11.3
250	100	607.5	85.8	**2 000**	100	61.5	16
				2 500	100	79.8	27.9
Pattern 2							
100	100	606.2	18.9	**100**	100	58.3	0.4
200	100	605.9	80.8	**1 000**	100	57.9	14
250	100	605.8	88.6	**2 000**	100	66.4	15.8
				2 500	100	103.3	29.2
Pattern 3							
100	100	606.6	28.7	**100**	100	58.3	1.2
200	88.8	605.9	96.4	**1 000**	100	58.9	15.7
250	73.5	603.6	95.8	**1 500**	100	60.6	15.4
				2 500	100	70.9	29.7
Pattern 4							
100	100	605	31.8	**100**	100	66.9	1
200	83.7	605.3	95.2	**1 000**	100	179.5	13.4
250	67.7	606	96.3	**1 500**	100	243.8	17.5
				2 500	53.9	265.6	33.2
Pattern 5							
100	100	606.8	20	**100**	100	70.8	1.5
200	80.5	604.9	95.4	**1 000**	100	199.1	15.1
250	69.2	605.2	96.5	**1 500**	100	250.5	16.4
				2 500	37.3	266.6	32.2
Pattern 6							
100	100	608.4	21.2	**100**	100	70.4	4.2
200	83.9	605.8	96.3	**1 000**	100	200.6	12.4
250	67.1	606.7	95.8	**1 500**	100	250.7	12.9
				2 500	40.3	268.3	31.4

As we can see in the left-hand side of Table VII, the CEP-ESB architecture is able to deal with an incoming rate of 100 events per second for all patterns. An incoming event rate of 200 events per second is also processed correctly in the case of event patterns 1 and 2; however, the rest of the patterns present problems to process all the data at that input ratio. Finally, at 250 events per second input rate, only the first pattern can correctly process the data, while the others cannot, showing a clear CPU overload and high memory consumption.

Concerning the CEP-Dataflow implementation, as shown in the right-hand side of Table VII, event patterns 1, 2, and 3 are processed correctly at 1000, 2000, and 2500 events per second/s of input rate, although the CPU overhead increases as the incoming message rate rises. In the case of patterns 3, 5, and 6, almost 50% of received events cannot be processed when the rate of 250 events per second/s is reached.

VI. RELATED WORK

First, it is important to highlight that we have not seen any related work that provides a performance comparison between a system where CEP is integrated with an ESB and one that integrates CEP with FBP. This related work section aims to describe work that attempts to process streaming data using FBP and CEP, which motivates the need for this comparative

performance study. We would also like to emphasize that when examining related work, we have noted that most FBP-based approaches focused on processing data from the IoT integrate MQTT brokers with the Node-RED tool [2], [29]–[31]. We consider that these approaches may be sufficient in some case studies, but not in others, since Node-RED provides a much more succinct syntax for defining event patterns and less performance than the Esper CEP engine. In any case, in the following lines, we examine the most relevant proposals using an FBP approach in conjunction with CEP.

Mahapatra [29] proposed a mashup tool called aFlux, which is based on FBP, for composing data analytics applications in a graphical way. This tool tries to combine the simplicity of use provided by IoT mashup tools and the flexibility brought by big data frameworks, such as Flink and Spark Streaming. This tool supports the following Flink components: CEP begin, CEP new pattern, CEP add condition, and CEP end. Although aFlux allows for the definition of event patterns, the number of types of pattern operators, data windows, and aggregation functions is much lower than the one provided by Esper CEP and therefore, the ability to detect situations of interest is also much lower.

Hung *et al.* [32] proposed an application framework that allows developers to build embedded and mobile applications by using an FBP paradigm. Particularly, they use JavaFBP and the Android operating environment. This work makes use of a machine learning tool called Mobile Weka to analyze data in real time; such a tool requires training a mathematical model with a previously stored database and a machine learning algorithm. In contrast to this approach, the integration of CEP with dataflows or with an ESB can analyze and correlate data *on the fly*, thanks to the use of the CEP engine, which could indeed be integrated with machine learning algorithms if needed, as previously done in previous works [2], [7]. In addition, CEP-Dataflow supports considerably more input and output data adapters compared to the ones proposed by Hung *et al.*'s work.

Young *et al.* [33] described an architecture for dynamic distributed data through an FBP model. This architecture allows us to coordinate data mining tasks between edge and cloud containers. Its implementation is done with Apache NiFi, which provides the user with the ability to create a real-time dataflow by connecting different processors in a graphical way. This work makes use of Anaconda, a Python-based data science platform, together with machine learning libraries to analyze data. As Hung *et al.*'s work, this one also does not integrate CEP technology with the FBP paradigm to analyze and correlate huge amounts of data in real time, so it requires the access and management of local databases, which can impact on system performance.

VII. Discussion and Conclusion

In response to the RQ posed in Section I, we can assert that as anticipated, a system based on the CEP-Dataflow implementation can be used to promptly detect situations of interest from incoming events in streaming in real time. However, it should be noted that we have found a series of limitations that can hinder the implementation of those systems, which require diverse processing flows of incoming data or to feed back the system with the complex events detected due to a more complex routing procedure. In addition, the number of input/output connectors provided by the CEP-Dataflow implementation is very limited and the implementation of new connectors by the developer is rather complex, a fact that is aggravated by the lack of documentation for the development of these systems.

Regarding the evaluation results, we have checked through the performance tests conducted that the CEP-Dataflow implementation offers better performance than the CEP-ESB one for all the defined benchmark patterns and all tested input message ratios. We consider that the ratio and pattern variety tested allow us to affirm that, in a generalized way, a CEP-Dataflow implementation offers better performance than the CEP-ESB one and performs better under situations of stress. This is undoubtedly due to the removal of the need to use an event bus. Without a doubt, when we choose the CEP-ESB option, we have to integrate 2 solutions (the ESB and the CEP engine), which entails information flows between one and the other and can lead to higher latency than with CEP-Dataflow. Furthermore, according to the Esper documentation, we also gain performance by avoiding the need to match events to statements and avoiding wrapping the objects to Bean at runtime [16]. Particularly, with the Raspberry Pi, the processing capacity is excessively reduced in the case of the CEP-ESB implementation. We have to bear in mind that an ESB requires a high amount of resources, as we have seen in Section III C.

As a conclusion, we can affirm that the use of CEP in the field of real-time streaming data processing can support intelligent decision making in a wide range of domains. Although the implementation can be done with two different alternatives (CEP-Dataflow and CEP-ESB), it is necessary to evaluate the advantages and disadvantages that each alternative may imply according to the needs of the system in question. On the one hand, it can be programmed in a traditional way, integrating the CEP engine into an ESB, where its efficiency in SOAs 2.0 for IoT scenarios has been proven. However, if the scenario requires better performance, it can be alternatively programmed through dataflow-based CEP if the system to be implemented has certain flexibility with the communication protocols and the workflows in the system since, as previously stated in Section III-C, this implementation is more limited with regards to input/output connectors, flows with complicated routing, as well as when changes in system flows are required and it is not intended to stop the execution of the system.

Acknowledgment

Details of the acknowledgements for the DECISION/P20_00865 project. Grant programme for R&D&I projects, aimed at universities and public research entities qualified as agents of the Andalusian Knowledge System, within the scope of the Andalusian Plan for Research, Development and Innovation (PAIDI 2020). Project 80% co-financed by the European Union, within the framework of the Andalusia ERDF Operational Programme 2014–2020 "Smart growth: An economy based on knowledge and innovation." Project funded by the Department of Economic Transformation, Industry, Knowledge and Universities of the Regional Government of Andalusia.

REFERENCES

[1] S. D. Liang, "Smart and fast data processing for deep learning in Internet of Things: Less is more," *IEEE Internet Things J.*, vol. 6, no. 4, pp. 5981–5989, Aug. 2019, doi: 10.1109/JIOT.2018.2864579.

[2] A. Akbar, A. Khan, F. Carrez, and K. Moessner, "Predictive analytics for complex IoT data streams," *IEEE Internet Things J.*, vol. 4, no. 5, pp. 1571–1582, Oct. 2017, doi: 10.1109/JIOT.2017.2712672.

[3] R. Mayer, B. Koldehofe, and K. Rothermel, "Predictable low-latency event detection with parallel complex event processing," *IEEE Internet Things J.*, vol. 2, no. 4, pp. 274–286, Aug. 2015, doi: 10.1109/JIOT.2015.2397316.

[4] A. García-de-Prado, G. Ortiz, and J. Boubeta-Puig, "CARED-SOA: A context-aware event-driven service-oriented architecture," *IEEE Access*, vol. 5, pp. 4646–4663, 2017, doi: 10.1109/ACCESS.2017.2679338.

[5] A. Garcia-de-Prado, G. Ortiz, and J. Boubeta-Puig, "COLLECT: COLLaborativE ConText-aware service oriented architecture for intelligent decision-making in the Internet of Things," *Expert Syst. Appl.*, vol. 85, pp. 231–248, Nov. 2017, doi: 10.1016/j.eswa.2017.05.034.

[6] K. C. Serdaroglu and S. Baydere, "An efficient multi-priority data packet traffic scheduling approach for fog of things," *IEEE Internet Things J.*, early access, May 27, 2021, doi: 10.1109/JIOT.2021.3084502.

[7] J. Roldán, J. Boubeta-Puig, J. Luis Martínez, and G. Ortiz, "Integrating complex event processing and machine learning: An intelligent architecture for detecting IoT security attacks," *Expert Syst. Appl.*, vol. 149, Jul. 2020, Art. no. 113251, doi: 10.1016/j.eswa.2020.113251.

[8] M. O. Gökalp, A. Koçyiğit, and P. E. Eren, "A visual programming framework for distributed Internet of Things centric complex event processing," *Comput. Elect. Eng.*, vol. 74, pp. 581–604, Mar. 2019, doi: 10.1016/j.compeleceng.2018.02.007.

[9] D. Corral-Plaza, G. Ortiz, I. Medina-Bulo, and J. Boubeta-Puig, "MEdit4CEP-SP: A model-driven solution to improve decision-making through user-friendly management and real-time processing of heterogeneous data streams," *Knowl. Based Syst.*, vol. 213, Feb. 2021, Art. no. 106682, doi: 10.1016/j.knosys.2020.106682.

[10] A. M. Rahmani, Z. Babaei, and A. Souri, "Event-driven IoT architecture for data analysis of reliable healthcare application using complex event processing," *Clust. Comput.*, vol. 24, no. 2, pp. 1347–1360, Jun. 2021, doi: 10.1007/s10586-020-03189-w.

[11] G. Mondragón-Ruiz, A. Tenorio-Trigoso, M. Castillo-Cara, B. Caminero, and C. Carrión, "An experimental study of fog and cloud computing in CEP-based real-time IoT applications," *J. Cloud Comput.*, vol. 10, no. 1, p. 32, Dec. 2021, doi: 10.1186/s13677-021-00245-7.

[12] A. Massaro, S. Selicato, R. Miraglia, A. Panarese, A. Calicchio, and A. Galiano, "Production optimization monitoring system implementing artificial intelligence and big data," in *Proc. IEEE Int. Workshop Metrol. Ind. 4.0 IoT*, Roma, Italy, Jun. 2020, pp. 570–575, doi: 10.1109/MetroInd4.0IoT48571.2020.9138198.

[13] O. Aziz, M. S. Farooq, A. Abid, R. Saher, and N. Aslam, "Research trends in enterprise service bus (ESB) applications: A systematic mapping study," *IEEE Access*, vol. 8, pp. 31180–31197, 2020, doi: 10.1109/ACCESS.2020.2972195.

[14] H. Derhamy, J. Eliasson, and J. Delsing, "IoT interoperability—On-demand and low latency transparent multiprotocol translator," *IEEE Internet Things J.*, vol. 4, no. 5, pp. 1754–1763, Oct. 2017, doi: 10.1109/JIOT.2017.2697718.

[15] S. Trabesinger, R. Pichler, D. Schall, and R. Gfrerer, "Connectivity as a prior challenge in establishing CPPS on basis of heterogeneous IT-software environments," *Procedia Manuf.*, vol. 31, pp. 370–376, Jan. 2019, doi: 10.1016/j.promfg.2019.03.058.

[16] "Esper Reference. Version 8.4.0. Chapter 21. EPL Reference: Data Flow." EsperTech. 2020. [Online]. Available: http://esper.espertech.com/release-8.4.0/reference-esper/html/dataflow.html (Accessed: Sep. 20, 2021).

[17] "Esper." EsperTech. 2021. [Online]. Available: http://www.espertech.com/esper/ (Accessed: Sep. 20, 2021).

[18] D. C. Luckham, *Event Processing for Business: Organizing the Real-Time Enterprise*. Hoboken, NJ, USA: Wiley, 2012.

[19] C. Inzinger, W. Hummer, B. Satzger, P. Leitner, and S. Dustdar, "Generic event-based monitoring and adaptation methodology for heterogeneous distributed systems," *Softw. Pract. Exp.*, vol. 44, no. 7, pp. 805–822, Jul. 2014, doi: 10.1002/spe.2254.

[20] "Esper Reference. Version 8.4.0. Chapter 5. EPL Reference: Clauses." EsperTech. 2020. [Online]. Available: http://esper.espertech.com/release-8.4.0/reference-esper/html/epl_clauses.html (Accessed: Sep. 20, 2021).

[21] M. Papazoglou, *Web Services and SOA: Principles and Technology*, 2nd ed. Essex, U.K.: Pearson Educ., 2012. [Online]. Available: http://catalogue.pearsoned.co.uk/educator/product/Web-Services-and-SOA-Principles-and-Technology/9780273732167

[22] P. Krill. "Make Way for SOA 2.0." InfoWorld. May 2006. [Online]. Available: https://www.infoworld.com/article/2654672/make-way-for-soa-2-0.html (Accessed: Sep. 20, 2021).

[23] O. Etzion *et al.*, "Panel session 3: Event-driven architectures and complex event processing," in *Proc. IEEE Int. Conf. Web Serv. (ICWS)*, Chicago, IL, USA, Sep. 2006, p. 36, doi: 10.1109/ICWS.2006.100.

[24] M. P. Papazoglou and W.-J. V. D. Heuvel, "Service-oriented design and development methodology," *Int. J. Web Eng. Technol.*, vol. 2, no. 4, pp. 412–442, Jul. 2006, doi: 10.1504/IJWET.2006.010423.

[25] R. Young, S. Fallon, P. Jacob, and D. O. Dwyer, "A flow based architecture for efficient distribution of vehicular information in smart cities," in *Proc. 6th Int. Conf. Internet Things Syst. Manage. Security (IOTSMS)*, Granada, Spain, Oct. 2019, pp. 93–98, doi: 10.1109/IOTSMS48152.2019.8939233.

[26] B. Zarrin and H. Baumeister, "Towards separation of concerns in flow-based programming," in *Proc. Companion 14th Int. Conf. Modularity (MODULARITY Companion)*, Fort Collins, CO, USA, 2015, pp. 58–63, doi: 10.1145/2735386.2736752.

[27] "Hardware and Software Requirements | MuleSoft Documentation." MuleSoft LLC. [Online]. Available: https://docs.mulesoft.com/mule-runtime/4.3/hardware-and-software-requirements (Accessed: Sep. 20, 2021).

[28] "Esper FAQ.What is the Footprint of Esper in a Typical Installation, i.e., What is the RAM, Disk and CPU Usage?" EsperTech. 2021. [Online]. Available: https://www.espertech.com/esper/esper-faq/#resource-consumption (Accessed: Sep. 20, 2021).

[29] T. Mahapatra, "Composing high-level stream processing pipelines," *J. Big Data*, vol. 7, no. 1, p. 81, Sep. 2020, doi: 10.1186/s40537-020-00353-2.

[30] T. Szydlo, R. Brzoza-Woch, J. Sendorek, M. Windak, and C. Gniady, "Flow-based programming for IoT leveraging fog computing," in *Proc. IEEE 26th Int. Conf. Enabling Technol. Infrastruct. Collab. Enterprises (WETICE)*, Poznan, Poland, Jun. 2017, pp. 74–79, doi: 10.1109/WETICE.2017.17.

[31] Z. Li, Y. Xiao, S. Liang, and S. Wang, "Design of smart home management system based on MQTT and FBP," in *Proc. Chin. Autom. Congr. (CAC)*, Xi'an, China, Nov. 2018, pp. 3086–3091, doi: 10.1109/CAC.2018.8623113.

[32] S.-H. Hung *et al.*, "MobileFBP: Designing portable reconfigurable applications for heterogeneous systems," *J. Syst. Archit.*, vol. 60, no. 1, pp. 40–51, Jan. 2014, doi: 10.1016/j.sysarc.2013.11.009.

[33] R. Young, S. Fallon, and P. Jacob, "Dynamic collaboration of centralized & edge processing for coordinated data management in an IoT paradigm," in *Proc. IEEE 32nd Int. Conf. Adv. Inf. Netw. Appl. (AINA)*, Krakow, Poland, May 2018, pp. 694–701, doi: 10.1109/AINA.2018.00105.

Guadalupe Ortiz was born in Madrid, Spain, in 1977. She received the Ph.D. degree in computer science from the University of Extremadura, Cáceres, Spain, in 2007.

She was as Assistant Professor with the University of Extremadura, since 2001. In 2009, she joined the University of Cadiz, Cádiz, Spain, as Professor of Computer Science and Engineering. She has published over 100 peer-reviewed papers in international journals, workshops, and conferences. Her research interests embrace service context awareness and their adaptation to mobile devices, as well as the integration of CEP in service-oriented architectures in the scope of the IoT and smart cities.

Iván Castillo was born in Puerto Real, Spain, in 1998. He received the B.S. degree in computer science and engineering from the University of Cadiz, Cádiz, Spain, in 2020.

He was an intern with the Department of Computer Science and Engineering, University of Cadiz, from 2019 to 2020. His research in that period was focused on the area of CEP for IoT applications through a microservice-oriented architecture perspective.

Alfonso Garcia-de-Prado was born in Madrid, Spain, in 1972. He received the Ph.D. degree in computer science and engineering from the University of Cadiz, Cádiz, Spain, in 2017.

For several years, he has been a Programmer, an Analyst, and a Consultant for various international industry partners. Since 2011, he has been an Assistant Professor with the University of Cadiz. His research focuses on trending topics, such as the CEP integration in service-oriented architectures and context awareness in the IoT.

Juan Boubeta-Puig was born in Cádiz, Spain, in 1985. He received the Ph.D. degree in computer science from the University of Cadiz, Cádiz, Spain, in 2014.

He is a tenured Associate Professor with the Department of Computer Science and Engineering, University of Cadiz. His research interests include real-time big data analytics through CEP, event-driven service-oriented architecture, IoT, blockchain and model-driven development of advanced user interfaces, and their application to smart cities, industry 4.0, e-health, and cybersecurity.

Dr. Boubeta-Puig was honored with the Extraordinary Ph.D. Award from UCA and the Best Ph.D. Thesis Award from the Spanish Society of Software Engineering and Software Development Technologies.