EventScala

A Type-Safe, Distributed and Quality-of-Service-oriented Approach to Event Processing
Bachelor-Thesis von Lucas Konstantin Bärenfänger
23. März 2017

1. Gutachten: Prof. Dr. Guido Salvaneschi
2. Gutachten: Pascal Weisenburger, M.Sc.

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik
Software Technology Group
Reactive Programming Technology

EventScala
A Type-Safe, Distributed and Quality-of-Service-oriented Approach to Event Processing

Vorgelegte Bachelor-Thesis von Lucas Konstantin Bärenfänger

1. Gutachten: Prof. Dr. Guido Salvaneschi
2. Gutachten: Pascal Weisenburger, M.Sc.

23. März 2017

Erklärung zur Bachelor-Thesis

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, 23. März 2017

(Lucas Konstantin Bärenfänger)

Contents

1 Abstract

Many of today's software systems rely on event processing (EP) for analyzing and reacting to events in real-time, including business-critical applications.

However, current EP solutions suffer from shortcomings, of which the following three are subject of this thesis. For one, EP queries are commonly expressed in the form of strings, allowing for no syntax and type checking at compile-time. Furthermore, many solutions employ a non-distributed, i.e., centralized, approach. Lastly, there appears to be no solution that enables queries to be annotated with quality of service (QoS) requirements.

The contribution of this thesis, the EventScala framework, sets out to explore ways to tackle these issues. To this end, EventScala features a domain-specific language (DSL) for expressing queries in a type-safe, statically checked fashion. Moreover, to achieve distributed execution, EventScala's execution engine runs queries by mapping their primitives and operators to concurrently executing nodes, which form a graph that resembles the structure of the given query. Finally, in EventScala, the annotation of queries with QoS requirements is supported at both the language and the execution level.

EventScala is based on the Scala technology stack. The DSL leverages the flexibility and extensibility of the Scala programming language. The execution engine leverages the Akka toolkit's abstractions for concurrent programming.

2 Introduction

Event processing (EP) is now the “paradigm of choice” [1] for building applications that analyze and react to events as they happen. In the corporate world, it is used in a variety of applications, e.g., business process management [2]. EP technology is also deeply embedded in the financial sector.

Nevertheless, current EP solutions still suffer from shortcomings, three of which are addressed in this thesis. For one, many EP solutions, e.g., Esper [3], receive queries in the form of strings, just like it is common in the field of relational databases. As a consequence, syntactically malformed or ill-typed queries are not detected at compile-time, allowing for them to fail at run-time. Furthermore, most if not all industry-grade EP solutions are non-distributed [4], i.e., centralized, solutions. Therefore, they can neither leverage parallelism to meet efficiency demands nor do they cope with the distributed nature of event sources. Lastly, while some research has been conducted on quality of service (QoS) in the EP domain (section 3.3), there appears to be no solution enabling queries to be explicitly annotated with QoS requirements. However—especially in the case of real-time applications that make decisions—whether or not a query’s requirements have been met during its execution might be crucial for the value and/or correctness of its result.

This thesis presents a type-safe, distributed and QoS-oriented approach to EP. As such, it explores ways to tackle the aforementioned issues. Its contribution, the EventScala framework, features a domain-specific language (DSL) for expressing EP queries as well as an execution engine for running them. The DSL enables to express queries in a type-safe manner, thus, serves as a static shield against malformed or ill-typed queries. It also allows for QoS requirements to be defined over queries in a fine-grained fashion, i.e., they can be defined over a query or its arbitrarily deeply nested subqueries. The execution engine enables queries to be run in a distributed fashion by mapping the primitives and operators of a given query to concurrently executing processing nodes, which form a graph that resembles the structure of said query. Each processing node continuously monitors its performance and triggers user-defined action whenever a QoS requirement is not met.

EventScala is based on the technology stack of the Scala ecosystem. Due to its flexible and extensible nature, the Scala programming language is known for its support for building DSLs. Therefore, the DSL is embedded into Scala, leveraging some of the features that make Scala a great host language, e.g., infix operator syntax and implicit conversions. The execution engine is built upon the abstractions provided by the Akka toolkit [5]. Akka’s approach to concurrency programming is based on the actor model [6], aiming to eliminate many of the difficulties commonly associated with the subject. Moreover, Akka appears to be especially suited for developing an EP solution, as it already embodies several of the characteristics common in the EP domain.

It can be stated that expressing queries using the DSL as opposed to SQL-like strings is a superior approach. One reason for this is that statically checked queries are far less likely to fail at run-time. Furthermore, running queries using the execution engine constitutes a way to benefit from the advantages of concurrent execution in the EP domain. Finally, this thesis shows that the concept of queries being annotated with QoS requirements can be incorporated at the language as well as at the execution level.

The remainder of this thesis is structured as follows. Related work is presented in section 3. The state of the art of EP in general is covered in section 3.1, its language-level integration is discussed in section 3.2, and sections 3.3 and 3.4 deal with distributed EP as well as QoS in the context of EP, respectively. Section 4 presents the EventScala framework, with section 4.1 introducing the underlying case class representation, section 4.2 covering the DSL, section 4.3 discussing the execution level and section 4.4 outlining the QoS approach. A simulation showcasing EventScala’s capabilities is presented in section 5. This thesis is concluded in section 6.

3 State of the Art

3.1 Event Processing (EP)

Event processing (EP) has, according to Hinze, Sachs and Buchmann [1], become the "paradigm of choice in many monitoring and reactive applications", with application scenarios including traffic monitoring, fraud detection, supply chain management and many more. (Refer to their paper "Event-based Applications and Enabling Technologies" [1] for an extensive list of use cases.)

This section introduces terminology as well as concepts common the EP field. In the end, some noteworthy EP solutions as well as the respective publications are referenced.

Chandy and Schulte [7] describe an event as "something that happens". Hinze, Sachs and Buchmann [1] introduce two more refined notions of events: "change events", e.g., an object changing its position, and "status events", e.g., a value yielded by a sensor.

After being observed and signaled, an event takes the form of an event instance, which corresponds to an event type. An event instance is commonly represented as a tuple of values, with the type of each element of the tuple being defined by the associated event type. For example, a temperature reading from a sensor "X" indicating 21 °C in temperature might be represented by the event instance `("X", 21)` with the event type being `(String, Int)`. For the sake of simplicity, the remainder of this thesis will refer to event instances as events.

Analogous to expressions in programming languages, which are either primitive values, e.g., `true` or `42`, or made up of other expressions that are combined by operators and/or functions, e.g., `true && isThisAGoodNumber(42)`, events may be primitive events or compositions/derivations of primitive and/or other composite/derived events. According to the definitions by Hinze, Sachs and Buchmann [1], "composite events" are "aggregations of events", whereas "derived events" are "caused by other events" and typically are "at a different level of abstraction". As an example of the latter, consider a series of failed login attempts might cause an intrusion event.

Etzion and Neblett [8] define an "event stream (or stream)" as a "set of associated events" that is "often" "temporally ordered". A stream solely consisting of events of the same type is called a "homogeneous stream"—as opposed to "heterogeneous".

Operators are defined over streams—as opposed to individual events. The `or` operator, for example, represents the union of two streams, and places the events of both streams in one result stream. The two streams the `or` operator takes as operands can be viewed as its incoming streams, the result stream can be viewed as its outgoing stream.

Traditionally, two approaches to EP can be distinguished:

Stream processing (SP) SP typically features operators that resemble those of relational algebra, e.g., `projection`, `selection`, `join`, etc. SP queries are usually expressed in some SQL dialect and constitute so-called "continuous queries". This term underlines an inversion of principles: In traditional database management systems (DBMSs), it is the data that is being persisted and not the queries. Continuous queries, however, are being persisted and run *continuous*ly, while it is the data that can be thought of as flowing through.

Complex event processing (CEP) CEP typically features operators that resemble those of boolean algebra, e.g., `and`, `or`, `not`. Operators such as `sequence` and `closure` are common, too. CEP queries are usually expressed using rule languages.

There is another significant difference between SP and CEP. As said, SP operators resemble those of relational algebra. Some relational operators (e.g., `join`), however, are blocking operators, i.e., blocking in the sense that they are defined over finite sets of data and block execution until these are available in their entirety. Streams, however, can be viewed as infinite sets of data. As a consequence, in SP, the respective operators are not applied to streams directly. Instead, they require their operand streams to be annotated with so-called windows, which are typically expressed "in terms of time or number of tuples [i.e., events]" [9]. At any point in time, a window only contains a finite number of the events of the respective stream. So-called consumption modes are the counterpart of windows in CEP. They are explained best through an example. The CEP operator `and`, for instance, is semantically ambiguous. The query `A and B` only specifies that an event of type `A` should be correlated with an event of type `B`. However, given the events `b1`, `b2`, `a1`, occurring in that order, it is not clear whether `a1` should be correlated with b1 or b2—this depends on the given consumption mode. (Chakravarthy et al. [10] further explain and formally specify consumption modes.) One of the differences between consumption modes and windows is that the former are applied to operators while the latter are applied to streams (i.e., operands). As pointed out in the book "Stream Data Processing: A Quality of Service Perspective" [9], consumption modes "can be loosely interpreted as load shedding, used from a semantics viewpoint rather than a QoS viewpoint", as they essentially dictate which events have to be kept in memory and which ones can be dropped as they will not be part of any correlation. It would be reasonable to state the same about windows.

To some degree, most current EP solutions do feature both SP and CEP operators. However, all of the EP solutions that were considered for this thesis treated CEP operators as second-class citizens. The SP engine Esper [3], to mention one example, features a typical SQL dialect, EPL (Event Processing Language), for expressing queries. Queries solely made up of CEP operators can be expressed using so-called `patterns`. These can then be used as operands of SP operators, as shown in listing 1. It is not possible, though, to use SP operators within a pattern, as shown in listing 2. Another solution, Flink [11], which considers itself to be a "stream processing framework", also features CEP operators, e.g., sequence (as `followedBy`), but does so in a designated library, called FlinkCEP [12].

Listing 1: In EPL, the `join` operator (`,`) can be applied to a `pattern`.

```
// 'lastEvent' forms a window, continuously containing the last event of the stream.
select * from
  Sensor1.std:lastEvent(), pattern[every (Sensor2 or Sensor3)].std:lastEvent()
```

Listing 2: On the contrary, in EPL, the `join` operator cannot be used within a `pattern`.

```
// Caution, invalid EPL!
select * from
  pattern[every (Sensor1 or (Sensor2.std:lastEvent(), Sensor3.std:lastEvent()))]
```

The distinction between SP and CEP can be considered blurry, as many books and publications often use the terms SP and CEP in their broad sense, meaning EP in general. SP and CEP do, however, pose different challenges when it comes to quality of service (QoS). (Section 3.4 covers their differences in terms of QoS.) Refer to section 5 "Analysis of Event vs. Stream Processing" of the paper "Events and Streams: Harnessing and Unleashing Their Synergy!" [13] for another comparison of SP and CEP.

The term event-driven architecture (EDA) stands for another important concept in the domain of EP. Chandy and Schulte [7] define it as the "concept of being event-driven", i.e., acting in response to an event, being applied to software. Furthermore, they identify following "five principles of EDA":

Individuality Each event is transmitted individually, not as part of a batch.

Push Events are pushed, not pulled/requested.

Immediacy Events are being reacted to immediately after reception.

One-way The type of communication is "fire-and-forget", events are neither being acknowledged nor replied to.

Free of command An event never prescribes the action that will be taken upon its reception.

Early EP solutions include HiPAC [14], SAMOS [15], Snoop [10] and SnoopIB [16]. These can give an impression of the developments from the late 1980s to the early 2000s. Up-to-date solutions are, for example, Esper [3] and Flink [11].

3.2 EP & Language Integration

This section starts out by examining and criticizing the way queries are typically expressed in the EP domain. Then, the notion of domain-specific languages (DSLs) is introduced. Lastly, it is described how DSLs are used to tackle problems common in the context of relational databases, as there are analogous problems to be solved in the context of EP.

As pointed out by Schilling et al. [4], “there does not exist any generally accepted definition language for [...] event processing”. However, many EP solutions—especially those that rely on dialects of SQL for expressing queries—generally receive those in the form of strings, just as traditionally done in DBMSs.

Leijen and Meijer [17] lay out why communicating SQL expressions in the form of “unstructured strings” is a bad approach. To begin with, “[p]rogrammers get no static safeguards against creating syntactically incorrect or ill-typed queries, which can lead to hard to find runtime errors”. Furthermore, it is noted that the programmer needs to know at least the respective SQL dialect as well as the “language that generates the queries and submits them”. To underline these very points, Kabanov and Raudjärv [18] present “a very simple example of an SQL query in Java” that contains multiple errors. The SQL query is embedded in Java as a `String`. Among other errors, it contains a misspelled SQL command. Furthermore, it is stated that the assumptions made about the type of the column and the result set could be wrong. As said, these problems are rooted in the practice of—to use the words of Spiewak and Zhao [19]—embeddeding queries “within application code in the form of raw character strings”. They point out that “[t]hese queries are unparsed and completely unchecked until run-time”. Thus, malformed queries do not cause the code they are embedded in to not compile but will cause trouble when being executed at run-time.

To tackle this issue in the domain of traditional DBMSs, the contribution of the paper by Spiewak and Zhao [19] is a so-called domain-specific language (DSL) named ScalaQL, which sets out “to make the host language compiler [i.e., scalac] aware of the query and capable of statically eliminating these runtime issues”. In fact, the contribution of the previously quoted paper by Leijen and Meijer [17] is very similar. Here, the host language is Haskell, while the problem domain is also relational databases. Leijen and Meijer seem to be the first ones to “show how to embed the terms and type system of another (domain-specific) programming language into the Haskell framework, which dynamically computes and executes programs written in the embedded language”.

To explain what DSLs are, it makes sense to turn to the book “DSLs in Action” by Ghosh [20], which appears to be the standard primer on the subject. It includes the following definition:

“A DSL is a programming language that’s targeted at a specific problem; other programming languages that you use are more general purpose. It contains the syntax and semantics that model concepts at the same level of abstraction that the problem domain offers.”

Furthermore, it is stated that—“[m]ore often than not”—DSLs are used by non-expert programmers. To enable them to do so, it is necessary that a DSL appeals to its target group by having “the appropriate level of abstraction”, i.e., it must embody the particular terminology, idioms, etc. of the respective domain. This, however, leads to what is called “limited expressivity”—“you can use a DSL to solve the problem of that particular domain only”. As an example of limited expressivity, the author mentions that while “UI designers feel comfortable writing HTML”, they cannot “build cargo management systems using only HTML”.

In his book, Ghosh [20] further differentiates the notion of DSLs. According to the author, there are internal as well as external DSLs. (Internal DSLs are also known as embedded DSLs.) He defines them as follows:

“An internal DSL is one that uses the infrastructure of an existing programming language (also called the host language) to build domain-specific semantics on top of it.”

“An external DSL is one that’s developed ground-up and has separate infrastructure for lexical analysis, parsing technologies, interpretation, compilation, and code generation.”

Leijen and Meijer [17] argue that internal DSLs “expressed in higher order, typed [...] languages” are better suited as a “framework for domain-specific abstractions” than external DSLs. This is, according to them, due to the fact that programmers only have to know the host language, since domain-specific abstractions are made available as “extension languages”. Other advantages mentioned include the possibility to leverage other “domain-specific libraries” as well as the possibility to use the host language’s infrastructure, e.g., its module and type system. Kabanov and Raudjärv [18] state more advantages of internal DSLs, e.g., “re-using the platform tooling, which in Java case [sic!] includes compilers, advanced IDEs, debuggers, profilers, and so on”. Furthermore, they claim that the development of an embedded DSL is much easier as it essentially “boils down to writing an API”.

Finally, it is to be noted that Scala lends itself very well as a host language of an internal DSL. For instance, Scala language features used in ScalaQL [19] include “operator overloading, implicit conversions, and controlled call-by-name semantics”. As ScalaQL essentially deprecates the string representation of relational database queries in the Scala ecosystem, it is the logical next step to also deprecate sting-based EP queries—by developing an internal DSL. (Obviously, ScalaQL is just one of many DSLs to do so, another approach worth mentioning would be Slick by Lightbend, Inc. [21].)

3.3 EP & Distributed Execution

In their paper "Distributed Heterogeneous Event Processing" [4], Schilling et al. argue that "distributing the handling of events" is of growing importance, if only due to "the emerging increase in event sources". It can be stated that other reasons for this trend are the need to exploit parallelism to meet increasing efficiency demands, avoiding single points of failure, and many more.

In fact, numerous approaches to distributed EP (e.g., [4, 22–29]) have been proposed in the academic world. However, these have—to the best of my knowledge—too few common characteristics to distill one reference model that represents academia's approach to distributed EP. Moreover, as pointed out by Schilling et al. [4], "there exists a gap between [...] academia and the industry", i.e., none of the approaches to distributed EP proposed by academics are actually used in industrial practice. One of the stated reasons for this is that EP technology used "in business applications" needs "to access context information related to business processes" that "often resides in centralized databases".

With neither a unified academic approach to distributed EP nor an industry-grade solution for it to show for, this section is confined to introduce distributed EP theoretically, i.e., through an abstraction. Later in this section, challenges that are present in this area of research are listed, along with references to publications that address these.

In their book "Event Processing in Action" [8], Etzion and Niblett introduce an abstraction called event processing network (EPN). An EPN is made up of processing elements. It is represented as a graph with the processing elements being the nodes of that graph. Nodes can have "input terminals" and/or "output terminals". Output terminals of one node may be connected to input terminals of other nodes. These connections form the edges of the graph. An edge shows "the flow of event instances through the network", thus, can be considered a stream. There are the following types of processing elements:

Event producers Event producers introduce events into the EPN.

Event consumers Event consumers receive events from processing elements of the EPN.

Event processing agents (EPAs) EPAs embody "three logical functions":

- Filtering, i.e., selecting events for further processing
- Matching, i.e., "[f]inding patterns among events" and creating the respective pattern events
- Derivation, i.e., deriving new events from the output of the pattern step

Global state elements Global state elements "represent stateful data that can be read by event processing agents when they do their work".

Channels Channels may be used instead of edges for connecting processing elements, as the behavior of channels may be defined explicitly.

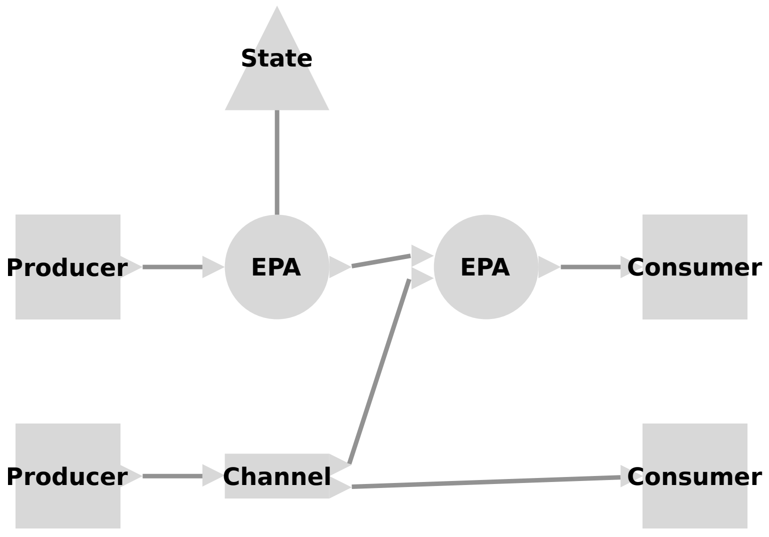
Figure 1 depicts an exemplary EPN.

It is to be stressed that an EPN diagram is merely an abstraction, comprised of "platform-independent definition elements". It does *not* assume a distributed implementation.

After the introduction of the EPN concept, Etzion and Niblett [8] lay out the "[i]mplementation perspective", in which the graph that the EPN is has to be materialized onto what are being called "runtime artifacts". This results in a "runtime system". Usually, there is no "one-to-one correspondence" between the processing elements of an EPN and the runtime artifacts of a respective implementation. With regard to this, the authors describe two "extremes":

- The entire EPN is be represented by one runtime artifact, resulting in one centralized runtime system.

Figure 1: An exemplary EPN, inspired by figure 2.7 on page 43 of the book "Event Processing in Action" [8]



- Each EPA is represented by one runtime artifact. These distribute events between each other and "can be placed on different server[s], allowing much of the [...] work to be performed in parallel."

The second scenario can be considered a prime example for distributed EP, even though there are obviously many ways to map EPAs to runtime artifacts, several of which might also be considered distributed approaches.

In the chapter "EPA assignment optimization", which is concerned with mapping "logical functions [i.e., EPAs] to physical runtime artifacts", Etzion and Niblett [8] mention parallel and distributed processing explicitly. Parallel processing is considered "[o]ne of the major ways to achieve [...] performance metrics". Then, three levels of parallelism are introduced, i.e., using multiple threads in one core, using multiple cores and using multiple machines. However, finding out "which activities should be run in parallel" is considered a "difficult" challenge. Regarding distributed processing, it is stated that "moving the processing close to the producers and consumers" constitutes an optimization method. "Partitioning", i.e., the practice of grouping EPAs that can "execute together [i.e., in parallel]", is introduced and considered "key" to both parallel and distributed execution. "Stratification", a partitioning approach in which EPAs are assigned to so-called "strata" is also described: "If `EPA1` produces events that are consumed by `EPA2`, then `EPA2` is placed in a higher stratum."

Regarding the aforementioned challenge of parallel execution, noteworthy approaches (e.g., [25–28]) have been proposed. The previously mentioned optimization technique of "moving the processing close to the producers and consumers" poses yet another challenge, especially when it comes to mobile producers/consumers. Interesting approaches to it (e.g., [23, 24, 29]) have been presented. Moreover, it appears that a good amount of the solutions to distributed EP proposed by academia are built upon loosely-coupled publish/subscribe systems (e.g., [22–24, 26]), which constitute a related field of research. Also, for the events of a stream to be put into time-based windows or for them to be temporally ordered, they need to be properly timestamped. Timestamping is a well-studied challenge in distributed systems in general and constitutes yet another related field of research. Many approaches have been proposed over time. Liebig, Cilia and Buchmann [22] present an interesting solution that is specifically aimed at distributed EP, leveraging a combination of NTP-synchronized local clocks and heartbeat events.

3.4 EP & Quality of Service

Appel, Sachs and Buchmann [30] note that "[f]uture software systems" have to be "responsive to events" in order to "adapt [...] software to enhance business processes". As an example, they mention software-controlled logistics processes that must be altered according to incoming updates on traffic. It is stressed that in such a scenario, in which critical business processes are triggered by events, "[t]he trust in such [...] systems depends to a large extent on the Quality of Service (QoS) provided by the underlying event system". The definition of QoS metrics as well as finding the means to monitor those is considered a "major challenge".

This section is structured as follows. At first, QoS metrics that have been identified for publish/subscribe systems—which do mostly apply to EP, too—are described. Afterwards, QoS metrics that are specific to SP or CEP are presented.

In their publication “Quality of Service in Event-based Systems”, Appel, Sachs and Buchmann [30] point out that “[f]or the [underlying] communication [...] the paradigm of choice is publish/subscribe”, i.e., many EP solutions are built upon publish/subscribe systems. Thus, they refer to a paper by Behnel, Fiege and Mühl [31], in which QoS metrics for publish/subscribe-based systems are identified. In this paper, publish/subscribe systems are defined as follows:

“The system model of the publish-subscribe communication paradigm is surprisingly simple. [... There are] three roles: publishers, subscribers and brokers. Publishers [...] provide information, advertise it and publish notifications about it. Subscribers [...] specify their interest and receive relevant information when it appears. Brokers mediate between the two by selecting the right subscribers for each published notification. [...] There can be a single centralized broker, a cluster of them or a distributed network of brokers.”

It is to be noted, however, that Appel, Sachs and Buchmann [30] explicitly recommend to refrain from considering generic QoS metrics. They recommend to compile a list of specific, i.e, non-generic, QoS requirements for the EP solution in question “rather than building a Swiss army knife EBS [i.e., event-based system] supporting all imaginable [...] QoS needs”. Nevertheless, for the sake of completeness, the following paragraphs summarize some of the generic QoS metrics presented by Behnel, Fiege and Mühl [31].

As said, even though the following QoS metrics were originally put together “in the context of distributed and decentralized publish-subscribe systems” [31], they do also apply to EP, since the underlying communication of many distributed EP solutions is based on the publish/subscribe paradigm. In order to interpret these QoS metrics in the sense of EP, one can simply think of an EPN’s event producers, event consumers and EPAs whenever publishers, subscribers and brokers of publish/subscribe systems are mentioned. Furthermore, statements about publishers and subscribers do not only apply to event producers and event consumers, respectively, but also to EPAs, as these do also consume and produce event streams. Notifications can be thought of as events. For example, a statement such as, say, “publishers annotate notifications they emit with priorities” can be interpreted as “event producers and EPAs annotate events they produce with priorities”. With no further ado, find below a summary of some of the QoS metrics proposed by Behnel, Fiege and Mühl [31].

Latency It is stated that “[s]ubscribers request a publisher that is within a maximum latency bound”. An “end-to-end latency” between a publisher and a subscriber “depends on the number of [...] hops between them” as well as on the time each broker takes to deal with a notification. However, in a distributed setting, “measured lower bounds” may at best “give hints” about whether some latency requirement can be met—not “absolute guarantees”. Preallocating paths is mentioned as a way to deal with latency requirements.

Bandwidth It is described that publishers announce upper/lower bounds regarding the stream they produce, whereas subscribers “restrict the maximum stream of notifications they want to receive”. It is recommended to consider bandwidth at the “per-broker” level. Given that “each broker knows the bandwidth it can make locally available to the infrastructure”, an upper-bound for an entire path can be estimated, allowing for routing “based on the highest free bandwidth”.

Message priorities It is proposed that publishers annotate the notations they produce with relative or absolute priorities, denoting their importance compared to other notifications produced either by themselves or elsewhere, respectively. Subscribers, on the other hand, specify priorities regarding their subscriptions. At the “per-broker” level, priorities “can be used to control the local queues of each broker”, resulting in the “end-to-end application” of the priorities along the entire path. This is, according to the authors, commonly implemented by letting notifications with higher priorities overtake those with lower priorities at each broker.

Delivery guarantees While subscribers announce “which notifications they must receive”, and where they are sensitive to duplicates, it is stated that publishers “specify if subscribers must receive certain notifications”. A simple approach that is mentioned is to simply let the system know which messages can be discarded. More sophisticated approaches concern “the completeness and duplication of delivery”, i.e., subscribers may receive notifications that are directed to them “at least once”, “at most once” or “exactly once”. While meshing is listed as a way to achieve “at least once”, it comes at the price of “increasing the message overhead”.

Disconnected subscribers are yet “[a]nother problem”, as they might stay disconnected, in which case “at least once” cannot be guaranteed, no matter how long the notifications are buffered.

Notification order It is explained that “[t]he order in which notifications arrive” is of significance in some cases while it is not in other cases. With centralized ordering, “ordering is [considered] easy to achieve”. However, “distributed ordering of events coming from different sources” is described as problematic. Deploying a “central broker to enforce a global ordering” is mentioned as a “generic approach” to tackle this challenge, imposing a “limit” on the “scalability of the overall infrastructure”, though.

Validity interval The importance of the infrastructure to know “how long a notification stays valid” is stressed. It is explained that this is specified either “in terms of time” or “by the arrival of later messages”. Using a specification based on “follow-up messages” to express that “only the most recent event is of interest” is mentioned as an example of the latter. This is called an “efficient approach”, as it allows for the infrastructure to “shorten its queues in high traffic situations”.

When interpreting the above list of QoS metrics in terms of EP, it becomes evident that these metrics can be applied to both SP as well as CEP. (As said, brokers are to be thought of as EPAs, and it has not been specfied whether these EPAs perform SP or CEP operations.) However, there are QoS metrics that specifically apply to either SP or CEP. Some of them have been identified in the paper “From Calls to Events: Architecting Future BPM Systems” [2] by Buchmann et al., several of which are summarized below.

SP-specific QoS metrics listed in section “QoS of Stream Processing” [2] include:

Timeliness One the one hand, it is stated that applications relying on SP “typically have timing constraints to meet”, thus, the timeliness of the “continuous processing of incoming events” is considered “one of the most relevant QoS requirements”. On the other hand, many of these applications are said to “tolerate approximate results”. According to the authors, these circumstances suggest a “common” “trade-off”, i.e., “accuracy for timeliness”. To the best of my understanding, an example of this trade-off would be favoring some less precise filter that therefore executes quickly over some more precise and more time-consuming filter.

Throughput With regards to timeliness, “achievable throughput” is considered a “closely related QoS metric”. SP solutions are said to “attempt to optimize” their “continuous query execution” in order to achieve maximum throughput. As load shedding is considered to “often” be the “only practical approach”, the result is, again, the “trade-off of accuracy for timeliness”.

Order Whether events may be processed out of order is listed as an “application dependent” “issue”.

CEP-specific QoS metrics listed in section “QoS of Event Composition” [2] include:

Order Establishing an “ordering between events” can be considered the most important QoS metric, as “achievable QoS [...] depends largely on the possibility” to be able to do so. As an example of an operator that requires events to be ordered, the `sequence` operator, “which is part of most [CEP] event algebras”, is mentioned. Furthermore, it is stated that the “natural ordering” is time-based, which, according to the authors, is no problem if “there is only one central clock” as well as only one event occurring per clock tick. If, however, there may be “multiple events” occurring at the same point in time, being “time-stamped by different clocks”, establishing a total order is said to be impossible. Lastly, it is explained that the granularity of timestamps plays a major role: Events that might have been distinguishable with fine-grained timestamps might no be distinguishable when being timestamped more coarsely.

Delay/loss of messages The delay or loss of messages is described as a “source of ambiguity”. As an example, it is explained that it is impossible to determine that an event “did not occur in a given interval” unless it can be asserted that the event in question is neither delayed nor lost. With regard to bounded networks, the authors refer to the 2g-precedence model as a possibility to tackle this challenge. With regard to unbounded networks, “such as the internet”, they refer to a paper called “Event composition in time-dependent distributed systems” [22]. (This paper also contains an explanation of the 2g-precendce model.)

4 EventScala

4.1 Case Class Representation

At the heart of the EventScala framework is the case class representation of queries. Thanks to it, EventScala's DSL and EventScala's execution engine can be thought of as separate modules that do not depend on each other in any way. On the one hand, using the DSL to express a query results in the case class representation of that query. On the other hand, running a query using the execution engine requires a case class representation of said query. However, a case class representation of a query does neither have to be obtained using the DSL nor it have to be executed using the execution engine. One could, for example, use EventScala's DSL to obtain the case class representation of a query and then generate a SQL-like string from it to execute it using, say, Esper. Likewise, one could develop a different DSL or even type out the case class representation of a query by hand and then pass it to EventScala's execution engine to be run. Essentially, EventScala's case class representation is a type-safe and platform-independent way to encode queries for EP systems in Scala.

In this section, EventScala's case class representation is presented in great detail. To this end, the hierarchy and structure of the traits and case classes that make up the case class representation of queries is explained.

At the top of the hierarchy is the `Query` trait, which only specifies that extending classes have to have a field called `requirements`, representing a set of QoS requirements that were specified for the respective query. (Refer to the end of this section as well as to section 4.4 for more information on QoS requirements in EventScala.)

As described in section 3.1, events are commonly represented by tuples of values, i.e., an event consisting of n values would be represented by a n-tuple. In EventScala, an event consists of at least one element and at most six elements. Analogously, the `Query` trait is extended by the traits `Query1`, `Query2`, ..., `Query6`, each representing a query that results in a stream of events consisting of 1, 2, ..., 6 elements, respectively. The traits `Query1`, `Query2`, ..., `Query6` do not specify any additional fields. They do, however, take 1, 2, ..., 6 type parameters, respectively, specifying the types of the elements of the events of the respective streams they represent. `Query2[String, Int]` is, for example, the type of a query that results in a stream of events which consist of two elements: a `String` and an `Int`.

A query usually consists of nested applications of operators over primitives. For example, one might subscribe to a stream of events consisting of a single `Int`, resulting in a `Query1[Int]`, as well as to a stream of events consisting of two `Strings`, resulting in a `Query2[String, String]`. One might then apply the `join` operator to these two stream subscriptions, resulting in a `Query3[Int, String, String]`. Afterwards, one could decide to drop the second `String` using the `dropElem3` operator, resulting in a `Query2[Int, String]`. When picturing this query as a graph (as illustrated in figure 2), the `dropElem3` operator would be the root node, having one child node, i.e., the `join` operator, which, in turn, would have two child nodes, i.e., the two stream subscriptions, which would be represented by the leaves of the graph. (In fact, this graph precisely resembles the graph of concurrently executing nodes that would be created if the query were run by EventScala's execution engine.)

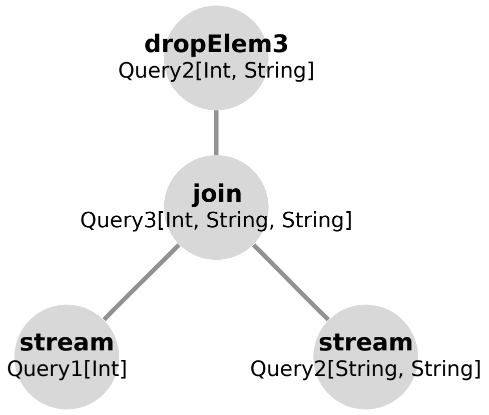
Analogously, the application of the `dropElem3` operator can be thought of as a unary query—unary in the sense that it has one subquery. Furthermore, the application of the `join` operator can be thought of as a binary query—binary in the sense that it has two subqueries. Lastly, the subscription to a stream can be thought of as a leaf query—leaf in the sense that it has no child queries. As a matter of fact, in EventScala, every query can be classified into being either a leaf, a unary or a binary query. Therefore, there exist three traits `LeafQuery`, `UnaryQuery` (specifying one field, `sq` (subquery), of type `Query`) and `BinaryQuery` (specifying two fields, `sq1` and `sq2`, both of type `Query`). All three traits extend the trait `Query`.

In the previous paragraph, it has been hinted that in EventScala, one might subscribe to streams or make use of the operators `join` or `dropElem3`. For the sake of completeness, find below the list of all primitives (i.e., leaf queries) and operators (i.e., unary and binary queries) available.

Leaf queries, i.e., traits extending the `LeafQuery` trait:

StreamQuery A `StreamQuery` expresses a subscription to a stream. The trait `StreamQuery` specifies one field of type `String`, `publisherName`, for the name of the publisher that is the source of the stream.

Figure 2: A conceptual depiction of a Query



SequenceQuery A `SequenceQuery` expresses a subscription to two streams with the CEP operator `sequence` being applied to them. The trait `SequenceQuery` specifies two fields of type `NStream`, `s1` and `s2`. A `NStream` is essentially a `Stream`, however, it is "Not a query", i.e., not extending the trait `Query`. If `NStream` would be a query, then `s1` and `s2` would represent the subqueries of a `SequenceQuery`, which would make it a binary query rather than a leaf query.

Unary queries, i.e., traits extending the `UnaryQuery` trait:

FilterQuery A `FilterQuery` expresses the application of the SP operator `where`. The `FilterQuery` trait specifies one field of type `Event => Boolean`, `cond`, for the filter predicate. The field for the subquery representing the operator's input stream is specified by the extending classes.

DropElemQuery A `DropElemQuery` expresses the application of the operator that EventScala offers instead of the EP operator `select`. While `select` lets one select which elements of the events of the operator's input stream to keep, the `dropElem` operator lets one specify which element to drop. The `DropElemQuery` trait specifies no fields. Which element of the events of the operator's input stream is to be dropped is specified through the extending classes' names, e.g., `DropElem1Of2`.

SelfJoinQuery A `SelfJoinQuery` expresses the application of the SP operator `join` but with one stream representing both of the operator's input streams. The `SelfJoinQuery` trait specifies two fields of type `Window`, `w1` and `w2`, for the windows that are applied to the operator's input streams. The field for the subquery representing both of the operator's input streams is specified by the extending classes.

Binary queries, i.e., traits extending the `BinaryQuery` trait:

JoinQuery A `JoinQuery` expresses the application of the SP operator `join`. The `JoinQuery` trait specifies two fields of type `Window`, `w1` and `w2`, for the windows that are applied to the operator's input streams. The fields for the two subqueries representing the operator's input streams are specified by the extending classes.

ConjunctionQuery A `ConjunctionQuery` expresses the application of the CEP operator `and`. The `ConjunctionQuery` trait specifies no fields. The fields for the two subqueries representing the operator's input streams are specified by the extending classes.

DinjunctionQuery A `DisjunctionQuery` expresses the application of the CEP operator `or`. The `DisjunctionQuery` trait specifies no fields. The fields for the two subqueries representing the operator's input streams are specified by the extending classes.

Up to this point, a hierarchy of traits but not one case class has been presented. The case class representation of a query is, however, made up of (possibly nested) case class instances. These case classes have the following in common:

- They extend exactly one of the traits `Query1`, `Query2`, ..., `Query6`, indicating the number and types of the elements of the events of the stream they represent.

- They extend exactly one of the traits StreamQuery, SequenceQuery, FilterQuery, DropElemQuery, SelfJoinQuery, JoinQuery, ConjunctionQuery or DisjunctionQuery, indicating what kind of primitive or operator application they represent.

For example, the case class Stream1[A] extends the trait Query1[A], indicating that it represents a stream of events consisting of 1 element of the generic type A, as well as the trait StreamQuery, indicating that it represents a subscription to a stream, i.e., a primitive. It has a field publisherName as specified by the trait StreamQuery as well as a field requirements as specified by the trait Query. Conjunction12[A, B, C], to provide another example, extends the trait Query3[A, B, C], indicating that it represents a stream of events consisting of 3 elements of the generic types A, B and C, respectively, as well as the trait ConjunctionQuery, indicating that it represents an application of the and operator. Obviously, it also has the field requirements.

More interestingly, though, the case class Conjunction12[A, B, C] also has two fields (sq1 of type Query1[A] and sq2 of type Query2[B, C]) for the subqueries representing the and operator's two input streams, which—against one's intuition—are not specified in the ConjunctionQuery trait. The reason for this is that while every case class extending ConjunctionQuery does have two fields named sq1 and sq2, the two fields' types are always different. For the case class Conjunction11[A, B], their types are Query1[A] and Query1[B], respectively, and for Conjunction21[A, B, C], to provide another example, they are Query2[A, B] and Query1[C], respectively. This hints at the biggest shortcoming of EventScala: As it is not possible to abstract over the length of type parameter lists in Scala, one cannot define one trait (like trait Query[A, B, ...]) for queries in general, but has to define one separate trait (e.g., Query1[A], Query2[A, B], and so forth) for queries representing streams of events consisting of one element, two elements, and so forth. (Tuples actually suffer from the same shortcoming: Tuple1[+T1], Tuple2[+T1, +T2], and so forth, are all defined separately in the Scala standard library.) To avoid a ridiculous amount of code repetition, in EventScala, events can consist of at most six elements. Still, this seemingly small number does not only call for the six separate traits Query1[A] to Query6[A, B, C, D, E, F], but also for 15 separate case classes extending the SequenceQuery trait as well as, to provide the most extreme example, for 36 case classes extending the DisjunctionQuery trait. Any future work on this project should aim to solve this issue, maybe by representing events and queries with HLists, which encode the types of their members (which can be Heterogenuous, i.e., of different types) in their type. They are provided by the popular Scala library Shapeless [32], a "type class and dependent type based generic programming library for Scala".

At this point, EventScala's case class representation has been explained to an extent that an example query in case class representation (listing 3) can be presented.

Listing 3: Case class representation of the query depicted in figure 2

```
val sampleQuery1: Query2[Int, String] =
  DropElem3Of3[Int, String, String](
    Join12[Int, String, String](
      Stream1[Int]("X", Set.empty),
      Stream2[String, String]("Y", Set.empty),
      SlidingTime(42),       // Sliding window of 42 seconds
      TumblingInstances(21), // Tumbling window of 21 events
      Set.empty),
    Set.empty)
```

The avid reader has certainly noticed that this is the case class representation of the query that has been informally described and illustrated as a graph (figure 2) previously in this section. It is to be stressed that this is a type-safe representation of said query. (This point will be discussed in detail in section 4.2.) Also, it is to be mentioned that it is also a platform-independent representation, i.e., it neither encodes data specific to the DSL that generated it nor does it encode data that is specific to the EP solution that will execute it.

Lastly, as the title of this thesis suggests, EventScala is a quality-of-service-oriented approach to EP. As such, it allows for QoS requirements, i.e., run-time requirements, to be expressed with each query. One might have noticed that sampleQuery1 (listing 3) contains the expression Set.empty four times. At these four points, it would have been possible to define a set of requirements (of type Set[Requirement]). When picturing the query as a graph once again, it becomes clear that requirements can be defined over every node of the graph. EventScala features two kinds of requirements, FrequencyRequirements and LatencyRequirements. (These will be discussed in greater detail in section 4.4.) It is to be noted, however, that the case classes representing these requirements are not platform-independent. This is due to the fact that they do not just contain the specification of the respective requirement. They also embody a callback closure that defines what to do whenever the respective requirement is not met during the execution of the query. The fact that this callback closure is being passed data about the processing node that is responsible for executing the respective (sub)query is what breaks platform independence. Therefore, queries containing QoS requirements are bound to be run by EventScala's execution engine. As there are—to the best of my knowledge—no other EP solutions that allow for the definition or execution of per-query QoS requirements, this does not constitute a real limitation. Listing 4 shows another example query, representing a stream subscription that is being applied to a filter, with the stream being required to emit at least 2 events per 5 seconds.

Listing 4: Case class representation of a query that specifies a QoS requirement

```
val sampleQuery2: Query2[Int, Boolean] =
  Filter2[Int, Boolean](
    Stream2[Int, Boolean](
      "Z",
      // If not at least 2 events are emitted every 5 seconds,
      // "Problem!" will be printed to the console.
      Set(FrequencyRequirement(GreaterEqual, 2, 5, _ => println("Problem!")))),
    // This filter predicate does not filter out any event. :)
    _ => true,
    // No QoS requirements are defined over the filter.
    Set.empty)
```

4.2 Domain-Specific Language

EventScala's DSL for expressing queries for EP systems sets out to achieve in the domain of EP what ScalaQL [19] and other DSLs achieved for relational databases, i.e., "statically eliminating [...] runtime issues" [19] caused by "syntactically incorrect or ill-typed queries" [17]. EventScala's DSL can be classified as an internal DSL. With EventScala being a Scala framework, its host language is obviously Scala.

This section is structured as followed. At first, the DSL is presented from a user's perspective, i.e., an overview of its features is given and advantages are pointed out. Then, notable parts of its implementation are discussed, e.g., the use of Scala features such as implicit conversion [33].

In the previous section, it has already been revealed which primitives and operators are supported by EventScala. In the listings 5, 6, 7 and 8, it will be shown how the DSL can be used to express queries made up of these primitives and operators, i.e., how the DSL can be used to express leaf, unary and binary queries.

Listing 5: Primitives, i.e., leaf queries

```
// Subscription to a stream
// of events of type ‘Int‘ from A
val stream1: Query1[Int] =
  // Here, the type has to be explicitly annotated:
  stream[Int]("A")

// Subscription to a stream
// of events of type ‘String, String‘ from B
val stream2: Query2[String, String] =
  stream[String, String]("B")

// Subscription to two streams
// of events of type ‘Int‘ and ‘Boolean‘, respectively,
// with the CEP operator ‘sequence‘ applied to them
val sequence1: Query2[Int, Boolean] =
  // Here, the types have to be explicitly annotated:
```

```
  sequence(
    nStream[Int]("C") ->
    nStream[Boolean]("D"))

// Subscription to two streams
// of events of type ‘Int, Int‘ and ‘Int‘, respectively,
// with the CEP operator ‘sequence‘ applied to them
val sequence2: Query3[Int, Int, Int] =
  sequence(
    nStream[Int, Int]("E") ->
    nStream[Int]("F"))
```

Listing 6: Application of unary operators, i.e., unary queries

```
// Application of the SP operator ‘where‘
// to ‘stream1‘ (‘Query1[Int]‘)
val filter1: Query1[Int] =
  stream1.where(_ % 2 == 0)
  // Alternatively: ‘stream1 where (_ % 2 == 0)‘

// Application of the SP operator ‘where‘
// to ‘sequence2‘ (‘Query3[Int, Int, Int]‘)
val filter2: Query3[Int, Int, Int] =
  sequence2.where((e1, _, e3) => e1 < e3)

// Application of the SP operator ‘dropElem2‘
// to ‘stream2‘ (‘Query2[String, String]‘)
val dropElem1: Query1[String] =
  stream2.dropElem2()
  // Alternatively: stream2 dropElem2()

// Application of the SP operator ‘dropElem1‘
// to ‘sequence1‘ (‘Query2[Int, Boolean]‘)
val dropElem2: Query1[Boolean] =
  sequence1.dropElem1()

// Application of the SP operator ‘selfJoin‘
// to ‘stream2‘ (‘Query2[String, String]‘)
val selfJoin1: Query4[String, String, String, String] =
  stream2.selfJoin(
    slidingWindow(42.instances), // Alternatively: ‘42 instances‘
    tumblingWindow(42.seconds))  // Alternatively: ‘42 seconds‘

// Application of the SP operator ‘selfJoin‘
// to ‘sequence1‘ (‘Query2[Int, Boolean]‘)
val selfJoin2: Query4[Int, Boolean, Int, Boolean] =
  sequence1.selfJoin(
    tumblingWindow(42.instances),
    slidingWindow(42.seconds))
```

Listing 7: Application of binary operators, i.e., binary queries

```
// Application of the SP operator ‘join‘
// to ‘stream1‘ (‘Query1[Int]‘)
// and ‘stream2‘ (‘Query2[String, String]‘)
val join1: Query3[Int, String, String] =
  stream1.join(
    stream2,
    slidingWindow(42.instances),
    tumblingWindow(42.seconds))

// Application of the SP operator ‘join‘
// to ‘sequence1‘ (‘Query2[Int, Boolean]‘)
// and ‘sequence2‘ (‘Query3[Int, Int, Int]‘)
val join2: Query5[Int, Boolean, Int, Int, Int] =
  sequence1.join(
    sequence2,
    slidingWindow(42.instances),
    tumblingWindow(42.seconds))

// Application of the CEP operator ‘and‘
// to ‘stream2‘ (‘Query2[String, String]‘)
// and ‘stream1‘ (‘Query1[Int]‘)
val conjunction1: Query3[String, String, Int] =
  stream2.and(stream1)
  // Alternatively: ‘stream2 and stream1‘

// Application of the CEP operator ‘and‘
// to ‘sequence1‘ (‘Query2[Int, Boolean]‘)
// and ‘sequence2‘ (‘Query3[Int, Int, Int]‘)
val conjunction2: Query5[Int, Boolean, Int, Int, Int] =
  sequence1.and(sequence2)
  // Alternatively: ‘sequence1 and sequence2‘

// Application of the CEP operator ‘or‘
// to ‘stream1‘ (‘Query1[Int]‘)
// and ‘stream2‘ (‘Query2[String, String]‘)
val disjunction1: Query2[Either[Int, String], Either[X, String]] =
  stream1.or(stream2)
  // Alternatively: ‘stream1 or stream2‘

// Application of the CEP operator ‘or‘
// to ‘sequence2‘ (‘Query3[Int, Int, Int]‘)
// and ‘sequence1‘ (‘Query2[Int, Boolean]‘)
val disjunction2: Query3[Either[Int, Int], Either[Int, Boolean], Either[Int, X]] =
  sequence2.or(sequence1)
  // Alternatively: ‘sequence1 or sequence2‘
```

Listing 8: Nested queries

```
// Nested application
// of 3 unary and 3 binary operators
// to 4 primitives
val nested1: Query3[Either[Int, String], Either[Int, X], Either[Float, X]] =
  stream[Int]("A")
    .join(
      stream[Int]("B"),
      slidingWindow(2.seconds),
      slidingWindow(2.seconds))
    .where(_ < _)
    .dropElem1()
    .selfJoin(
      tumblingWindow(1.instances),
      tumblingWindow(1.instances))
    .and(stream[Float]("C"))
    .or(stream[String]("D"))

// Nested application
// of 2 binary operators
// to 3 (!) primitives
val nested2: Query4[Int, Int, Float, String] =
  stream[Int]("A")
    .and(stream[Int]("B"))
    .join(
      sequence(
        nStream[Float]("C") ->
        nStream[String]("D")),
      slidingWindow(3.seconds),
      slidingWindow(3.seconds))
```

Furthermore, the section presenting the case class representation of queries also revealed that EventScala supports queries to be annotated with QoS requirements, i.e., `FrequencyRequirements` and `LatencyRequirements`. As already mentioned, every query, including arbitrarily deeply nested subqueries, can be annotated with an arbitrary amount of QoS requirements. In listing 9, it is demonstrated how requirements are expressed using the DSL. It is shown how two of the subqueries of the query `nested1` from listing 8 are annotated with requirements.

Listing 9: Nested queries that specify QoS requirements

```
val nested1WithQos: Query3[Either[Int, String], Either[Int, X], Either[Float, X]] =
  stream[Int]("A")
  .join(
    stream[Int]("B"),
    slidingWindow(2.seconds),
    slidingWindow(2.seconds))
  .where(_ < _)
  .dropElem1(
    // `LatencyRequirement`
    latency < timespan(1.milliseconds) otherwise { (nodeData) =>
      println(s"Events reach node `${nodeData.name}` too slowly!") })
  .selfJoin(
    tumblingWindow(1.instances),
    tumblingWindow(1.instances),
    // `FrequencyRequirement`s
    frequency > ratio( 3.instances,  5.seconds) otherwise { (nodeData) =>
      println(s"Node `${nodeData.name}` emits too few events!") },
    frequency < ratio(12.instances, 15.seconds) otherwise { (nodeData) =>
      println(s"Node `${nodeData.name}` emits too many events!") })
  .and(stream[Float]("C"))
  .or(stream[String]("D"))
```

At this point it becomes apparent why the definition of QoS requirements in EventScala's DSL (and, by extension, EventScala's case class representation) is platform-specific: A QoS requirement always defines a callback closure that expects to be invoked with data, i.e., `nodeData`, that is specific to EventScala's execution engine. The `nodeData` parameter can be explained as follows. Each of a query's subqueries (and, in turn, each of its subqueries) is mapped to one processing node when run by the execution engine. Whenever one of the requirements of such a query cannot be met during run-time, the callback closure of the respective requirement is called. This closure is invoked with run-time data about the processing node that executes the respective query. This run-time data is represented by an instance of the case class `NodeData`.

Expressing EP queries using EventScala's DSL has many advantages over expressing them in the form of unstructured strings, including the ones listed below:

Syntax Obvious but nevertheless very important is the fact that syntactically incorrect queries would fail to compile instead of causing run-time errors. For example, forgetting the dot before adding another method call to the chain or misspelling a method's name would cause a compile-time error.

Type-safety The type of a query encodes the number and types of the elements of the events of the stream represented by it, e.g., a query of type `Query2[Int, String]` represents a stream of events with two elements of type `Int` and `String`, respectively. This information is used to provide static safeguards against a variety of malformed queries, for example:

- Type-safety of the `filter` operator: The `where` method takes a Scala closure that represents the filter predicate. The number and types of the parameters of that closure match the number and types of the elements of the events of the stream that is represented by the respective query. For example, given a query of type `Query2[Int, String]`, the programmer has to supply a closure with exactly two parameters of type `Int` and `String`, respectively. By extension, it is guaranteed that within the closure's body, these parameters are treated as an `Int` and a `String`, respectively, e.g., calling the `String` method `capitalize` on the first parameter would cause a compile-time error.

- Type-safety of the `dropElem` operator: Which ones of the `dropElem` methods are available depends on the number of elements of the events of the stream represented by the query in question. For example, on a `Query1`, not one `dropElem` method can be invoked. Given a `Query2`, to provide another example, `dropElem1` and `dropElem2` can be called. Calling `dropElem3` on a `Query2`, however, results in a compile-time error.

- Type-safety of the operators `selfJoin`, `join` and `and`: EventScala supports up to six elements per event. Therefore, the application of the operators `selfJoin`, `join` and `and` is only possible if the resulting query represents a stream of events with at most six elements. For example, when calling the `join` method on a `Query4`, only a `Query1` or a `Query2` can be passed to it as the second operand of the `join` operator.

Tooling As the DSL is an internal DSL that is embedded into Scala, every query expressed in it is a Scala expression. As a consequence, IDEs and other tools from the Scala ecosystem can be leveraged when expressing queries using the DSL. For example, one could benefit from the static safeguards listed above without ever manually hitting the compile button, as IDEs such as JetBrains IntelliJ IDEA [34] continuously perform static analysis while typing.

Obviously, this list of advantages could have been presented earlier, i.e., as part of the section on EventScala's case class representation of queries. All of the listed benefits also apply to the case class representation, since the DSL can be viewed as syntactic sugar for it.

Finally, as the DSL's usage has been demonstrated and its advantages have been pointed out, the remainder of this section discusses how it is implemented, i.e., some of the leveraged patterns and Scala features are described.

In EventScala's DSL, operators are represented by methods. It might look as if these methods were defined on the traits `Query1`, `Query2`, ..., `Query6`. Assuming a value `q` of type `Query2`, one can, for example, write `q.dropElem1()`, which suggests that `dropElem1` is a method of `Query2`. This would, however, violate the separation of data and logic. Therefore, it is to be stressed that `Query2`—along with every trait or case class that is part of EventScala's case class representation—does not come with any logic whatsoever, i.e., no methods are defined or implemented.

In order to make method calls such as `q.dropElem1()` possible, the DSL heavily relies on one of Scala's advanced language features, implicit conversion. In a blog post called "Pimp my Library" [33], Martin Odersky, the original designer of the Scala programming language, explains this feature. In the context of this thesis, it is sufficient to understand the following use case: Whenever a value of type `X` is required but instead a value of type `Y` is being supplied, the compiler tries to find a function that is capable of converting the `Y` value to a `X` value. For this to work, the function has to be in scope, it has to be marked with the `implicit` keyword and its signature has to be `Y => X`. As an example, Odersky introduces `x` of type `Array[Int]` and assigns `x` to `v`, a variable of type `String`. Obviously, this would normally result in a compile-time error. However, it does not, since the function depicted in listing 10 is in scope.

Listing 10: Implicit conversion function [33]

```
implicit def array2string[T](x: Array[T]) = x.toString
```

At this point it becomes apparent why method calls such as `q.dropElem1()` do not result in a compile-time error: `q` is of type `Query2`, which neither defines nor implements the method `dropElem1`. However, there is a case class `Query2Helper` which does have this method. There also is an implicit conversion function that takes a `Query2` and returns a `Query2Helper`. Needless to say, for each trait `Query1`, `Query2`, ..., `Query6`, there is a case class `Query1Helper`, `Query2Helper`, ..., `Query6Helper`, respectively, as well as a respective implicit conversion function in place.

This is basically a simplification of an approach proposed by Debasish Ghosh in his book “DSLs in Action” [20]. In chapter 6.4 “Building a DSL that creates trades”, he uses a “[s]equence of implicit conversions” and the “subsequent creation of helper objects” to construct a tuple that represents something called a fixed income trade. This pattern can be described as follows: The DSL is used to construct a tuple. The tuple’s first element should be of type `A`, the second one of type `B`, the third one of type `C`, etc. Helper classes `AHelper`, `ABHelper`, `ABCHelper`, etc. as well as the respective implicit conversion functions with the signatures `A => AHelper`, `(A, B) => ABHelper`, `(A, B, C) => ABCHelper`, etc. are defined. `AHelper` has a `method1` that takes a `B` and returns an `(A, B)`, `ABHelper` has a `method2` that takes a `C` and returns an `(A, B, C)`, and so forth. This pattern allows for expressions such as `a method1 b method2 c /* ... */` (with `a` being an `A`, `b` being a `B`, and so forth). As said, Ghosh [20] presents this pattern along with an example DSL is based upon it. He demonstrates how the expression `200 discount_bonds IBM for_client NOMURA on NYSE at 72.ccy(USD)` can be used to obtain a tuple that represents a fixed income trade, i.e., `((72, USD), NYSE, NOMURA, 200, IBM)`.

Another Scala feature that the DSL heavily relies upon is operator overloading. Technically, however, Scala does not support operator overloading, as it does not even have operators. Instead, what seem to be operators are actually methods. The arithmetic operator + that is defined over two `Int`s, for instance, is really just the method `abstract def +(x: Int): Int` of the abstract class `Int`. As such, the two expressions `40 + 2` and `40.+(2)` are actually equivalent. With many characters being legal identifiers that can be used as method names, e.g., +, >, <, |, overloading an operator in Scala is nothing more than defining a method. This feature is used throughout the DSL. For example, in order to construct a `FrequencyRequirement`, one might write `frequency > ratio(/* ... */) /* ... */`. Calling the function `frequency` returns a case object called `FrequencyHelper`, which defines a bunch of methods, including >, >=, <, <=, etc., all of which take one argument of type `Ratio`.

Lastly, variable-length argument lists (also known as varargs) constitute yet another Scala feature that is used throughout the DSL. Varargs are, for instance, used to syntactically reflect the fact that queries in EventScala can be annotated with zero or more QoS requirements. Therefore, `Requirement*` is the type of the last parameter of every method of the DSL that returns a query. The `stream[A]` method, for example, specifies two parameters, i.e., `publisherName` of type `String` and `requirements` of type `Requirement*`. As a consequence, this method might be supplied with no requirement, e.g., `stream[Int]("X")`, with one requirement, e.g., `stream[Int]("X", r1)`, with two requirements, e.g., `stream[Int]("X", r1, r2)`, etc. (with `r1`, `r2`, etc. being of type `Requirement`). If the DSL would not rely on varargs but simply use `Set[Requirement]` as the type of the `requirements` parameter—as it is the case in the case class representation—these method calls would look much more verbose: `stream[Int]("X", Set.empty)`, `stream[Int]("X", Set(r1))`, `stream[Int]("X", Set(r1, r2))`, etc.

4.3 Execution Engine

EventScala does not only offer a type-safe case class representation of EP queries and a DSL to generate the former, it also offers a way to execute such queries. As the title of this thesis suggests, EventScala’s execution engine allows for running queries in a distributed fashion.

This section is structured as follows. The execution graph is based on the Scala toolkit Akka [5] and, as such, on the actor model [6]. Even though this is not the place to cover either of them in depth, the first part of this section will introduce Akka and the actor model to the extent necessary. Afterwards, it is described why Akka seems particularly suitable for use in the EP domain. Lastly, the implementation of the execution engine is described.

EventScala’s execution engine is implemented using the Scala framework Akka, which is based on the actor model. According to Akka’s official documentation [35], actors “give you” “high-level abstractions for distribution, concurrency and parallelism” as well as an “[a]synchronous, non-blocking and highly performant message-driven programming model”.

As stated in Akka's documentation, concurrency means "that two tasks are making progress", without preventing each other from doing so. They do not necessarily progress at the same time, though, i.e., "they might not be executing simultaneously". As a example, time slicing is mentioned. Parallelism is then described to be the case "when the execution can be truly simultaneous". Furthermore, a method call is described as being synchronous "if the caller cannot make progress until the method returns". A method call is considered asynchronous when it "allows the caller to progress" right after issuing the call. In this case, the completion of the method is signaled through, say, the invocation of a callback closure.

Using Akka (and, by extension, the actor model), applications are essentially built by creating hierarchies of actors that send and receive messages. According to Akka's documentation, actors are "container[s]" for state, behavior, a mailbox, child actors and a supervision strategy. These five characteristics are then described in detail. These descriptions from the documentation [35] are briefly summarized below.

State An actor is a stateful abstraction. Therefore, an actor typically contains variables reflecting "possible states the actor may be in", e.g., a variable representing a "set of listeners" or "pending requests". Most importantly, this data is safeguarded from "corruption by other actors". Furthermore, "conceptually", each actor is represented by its own thread, which is also "completely shielded from the rest of the system".

Behavior Whenever an actor processes a message, it triggers the actor to behave in a certain way. Behavior is described as a "function which defines the actions to be taken in reaction to the message".

Mailbox As it is "[a]n actor's purpose" to process messages, its mailbox, "which connects sender and receiver", deserves special attention. An actor's mailbox enqueues the messages directed to it "in the time-order of send operations". As a result of this, "messages sent from different actors may not have a defined order at runtime". On the other hand, if the mailbox only contains messages from one sender, they are ensured to be in the order in which they were sent.

Child actors Actors may create child actors which they will then "automatically supervise".

Supervision strategy An actor embodies a "strategy for handling faults of its children". (EventScala is a research project that does not concern itself with fault tolerance, therefore, this is not relevant in this context.)

It is to be stressed that while actors "are objects which encapsulate state and behavior", "they communicate exclusively by exchanging [immutable] messages" [35]. Therefore, the fields and methods defined by an actor cannot be accessed in a direct, synchronous fashion. Instead, an actor can only be interacted with asynchronously, i.e., through messages. This is key when it comes to how Akka tries to take the pain out of concurrent and asynchronous programming: The communication between actors through immutable messages does not assume any kind of shared memory, rendering error-prone primitives such as locks, etc. unnecessary. Furthermore, even though actors run concurrently, the code that is defined within each actor can be thought of as being executed sequentially.

In section 3.1, characteristics of EDAs [7] have been listed. Conveniently, these appear to directly apply to Akka and, by extension, the actor model, too. This is why Akka is such a natural fit for implementing an EP engine.

Individuality An actor sends out each message individually.

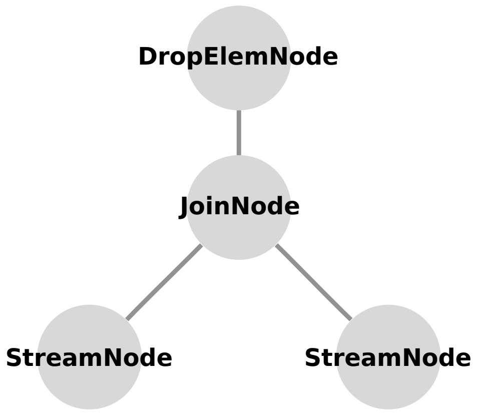
Push An actor sending out a message is never caused by some kind of request. (It could, however, be the reaction to some message previously received by the actor, which does not constitute a request, though, due to the last of the characteristics, i.e., "free of command".)

Immediacy An actor reacts to a message immediately after receiving it. (In fact, it could be stated that *all* it does is reacting to messages right after receiving them.)

One-way By default, an actor neither acknowledges nor replies to a received message. (Of course, this behavior can be explicitly implemented.)

Free of command An actor always decides how to react to a received message, said reaction is never dictated by the message.

Figure 3: Execution graph of the query defined in listing 3



EventScala's execution engine creates an execution graph (sometimes referred to simply as graph) for the query that is to be run. The execution graph consists of actors organized in a graph hierarchy. In section 4.1, it has been described how a query in case class representation could be pictured as a graph. It has already been pointed out then, that this conceptual graph directly corresponds to what the execution graph for that query would look like, i.e., it precisely resembles the hierarchy in which the actors of said graph would be assembled. In this conceptual graph, the outermost query has been depicted as the root node of the graph, its subquery as the child node of the root node, and so forth. In the execution graph of a query, these nodes are actors. Obviously, each actor represents either a primitive or an operator of a given query, i.e., either a `LeafQuery`, `UnaryQuery` or `BinaryQuery`. Actors representing primitives, i.e., `LeafQueries`, constitute the leaves of the graph. They receive messages, i.e., events, from so-called `Publishers`, which publish the events of the streams to which the primitives represent subscriptions. An actor representing a primitive then forwards these events to its parent actor, which represents an operator, i.e., either a `UnaryQuery` or a `BinaryQuery`. An actor representing an operator performs the respective SP or CEP operation on the incoming stream(s) and sends the events of the resulting stream up to its own parent actor. Finally, the actor representing the root of the graph invokes a closure for every event of its result stream. This closure is provided by the user and takes the respective event as argument.

As it would seem natural, the classes that make up the execution graph of a query structurally resemble the classes that make up its case class representation. Accordingly, corresponding to the `LeafQuery` trait there is a trait `LeafNode`, corresponding to `UnaryQuery` there is `UnaryNode` and corresponding to `BinaryQuery` there is `BinaryNode`. The traits `LeafNode`, `UnaryNode` and `BinaryNode` extend the `Node` trait, which, in turn, extends the `Actor` trait. As expected, the classes `StreamNode` and `SequenceNode` extend the `LeafNode` trait, `FilterNode`, `DropElemNode` and `SelfJoinNode` extend `UnaryNode` and `JoinNode`, `ConjuctionNode` and `DisjunctionNode` extend `BinaryNode`.

Figure 3 shows the execution graph that EventScala's execution engine would create for the query defined in listing 3.

Some classes, e.g., `JoinNode`, extend a trait called `EsperEngine`, while others, e.g., `FilterNode`, do not. The actors extending `EsperEngine` are essentially equipped with their own instance of the EP engine Esper [3], which they use to perform the respective operation they represent. This approach has two advantages. At first, by relying on Esper's implementation of more complex operators, e.g., `join`, a (possibly incorrect) implementation of such operators does not have to be provided. Moreover, resolving the semantic ambiguity of the `sequence` as well as the `and` operator is also taken care of by Esper's implementation. It is to be noted that some of the code of the `EsperEngine` trait was inspired by a Lightbend Activator Template called "CEP with Akka and Esper or Streams" [36].

Obviously, the execution graph of a query is created dynamically at run-time. At first, a `Node` corresponding to the outermost query of the respective query is created as a top-level actor of an `ActorSystem`. From then on, the graph builds up recursively, i.e., a `UnaryNode` will create a child actor that corresponds to the one subquery of the `UnaryQuery` which it represents. Likewise, a `BinaryNode` will create two child actors that correspond to the two subqueries of the `BinaryQuery` which it represents. `LeafNodes` will issue `Subscription(s)` to `Publisher(s)`. When a `Publisher` acknowledges a `LeafNode`'s subscription, the `LeafNode` either sends a `Created` message to its parent actor, or—if it is the root actor of the graph—invokes a closure that is to be called as soon as the graph is fully built (which supplied by the user). `UnaryNodes` and `BinaryNodes` also either send a `Created` message to their parent actors or call the aforementioned closure as soon as all of their child actors have sent them the `Created` message.

Finally, it is noteworthy that at run-time, events are represented by the classes `Event1`, `Event2`, ..., `Event6`, all of which extend the trait `Event`. `Publishers` pass up `Events` to the leaves of an execution graph and, within this graph, `Nodes` pass them up to their parent actors. Unlike their counterparts `Query1`, `Query2`, ..., `Query6`, `Events` do not have type parameters for each of their elements. Instead, the elements of an `Event` are of type `Any`. However, Akka actors are untyped anyway, i.e., actors receive any message and can send any message to any actor.

4.4 Quality of Service

As the title of this thesis suggests, EventScala represents a QoS-oriented approach to EP. As such, it allows not only for QoS requirements to be defined over queries but also for their enforcement at run-time. To this end, the execution graph that EventScala's execution engine creates for a given query features so-called QoS monitors. These surveil an actor's performance and carry out a user-defined reaction when a requirement is not met.

This section is structured as follows. At first, the notion of QoS monitors is introduced conceptually. Then, implementation details are laid out. Lastly, the two exemplary monitors that come with EventScala are described.

Conceptually, when picturing a query as a graph, QoS requirements can be defined over each node of that graph. To be more precise, an arbitrary amount of `FrequencyRequirements` and `LatencyRequirements` can be defined over each node of the graph. As this conceptual graph precisely resembles the execution graph of the same query, QoS requirements of a (sub)query are being dealt with by the actor that represents the (sub)query in the execution graph. Therefore, each actor has to monitor its performance, and, whenever a requirement is not being met, react to this circumstance by invoking a closure that has been specified along with the respective requirement. (To this end, both case classes `FrequencyRequirement` and `LatencyRequirement` specify a field named `callback`.) This closure might even carry out curative measures, as its arguments include the respective actor's `ActorContext`, which embodies run-time information about the actor.

Informally speaking, in order to create an execution graph for a given query, EventScala's execution engine requires the user to provide reference to one class representing a frequency monitor as well as to one class representing a latency monitor. At run-time, each actor of the execution graph gets exactly one instance of both of them—regardless of whether or not the query that the actor represents specifies any QoS requirements. As a consequence, at run-time, each actor has one frequency monitor instance as well as one latency monitor instance.

Technically speaking, EventScala's execution engine requires the user to provide two classes that extend the `MonitorFactory` trait, one for frequency monitoring and one for latency monitoring. A `MonitorFactory` can create instances of classes that extend one of the traits `LeafNodeMonitor`, `UnaryNodeMonitor` or `BinaryNodeMonitor`. Thus, at run-time, a `LeafNode` will have a `LeafNodeMonitor` for frequency monitoring as well another `LeafNodeMonitor` for latency monitoring. The former will have been created by the factory supplied for frequency monitoring and the latter will have been created by the factory supplied for latency monitoring. Analogously, the same is true for `UnaryNodes` and `BinaryNodes`.

EventScala has been designed to make it easy for users to implement their own monitors. To do so, one has to provide four classes that implement the four traits `LeafNodeMonitor`, `UnaryNodeMonitor`, `BinaryNodeMonitor` as well as `MonitorFactory`, respectively. EventScala comes with one exemplary frequency monitor as well as with one exemplary latency monitor. The examplary frequency monitor, for instance, is comprised of the following four classes:

- `AverageFrequencyLeafNodeMonitor` extending the trait `LeafNodeMonitor`
- `AverageFrequencyUnaryNodeMonitor` extending the trait `UnaryNodeMonitor`
- `AverageFrequencyBinaryNodeMonitor` extending the trait `BinaryNodeMonitor`
- `AverageFrequencyMonitorFactory` extending the trait `MonitorFactory`

However, the question remains how these monitors actually surveil the performance of the actor they are associated with. Each monitor implements so-called lifecycle methods, i.e., hooks into the respective actor's lifecycle. This approach has been inspired by React, Facebook's JavaScript library for building UIs [37], in which so-called components have so-called lifecycle methods "that you can override to run code at particular times in the process" [38]. `LeafNodeMonitor`, `UnaryNodeMonitor` and `BinaryNodeMonitor` all specify three methods that need to be implemented by extending classes:

- `onCreated`
- `onEventEmit`
- `onMessageReceive`

As the names of these methods suggest, the actors of an execution graph invoke them at certain times during their lifecycle. `onCreated` is being called only once, right after all of the actor's children have issued a `Created` message to it. `onEventEmit` is being invoked whenever the actor emits an event. `onMessageReceive` is being called whenever the actor receives a message that has not been generated by the execution graph, as it might have been sent by another monitor. (This way, communication between the monitors of the actors of a graph is facilitated.) Obviously, whenever `onEventEmit` is called, it is passed the respective event as an argument. Likewise, whenever `onMessageReceive` is called, it is passed the respective message. Furthermore, depending on the type of the respective actor, when being invoked, all three methods are always passed either an instance of the `LeafNodeData` case class, an instance of the `UnaryNodeData` case class or an instance of the `BinaryNodeData` case class. All of these contain the name of the respective actor as a `String` as well as its `ActorContext`. `UnaryNodeData` and `BinaryNodeData` also contain an `ActorRef`, i.e., a reference to an actor, for each of their child actors, i.e., child nodes.

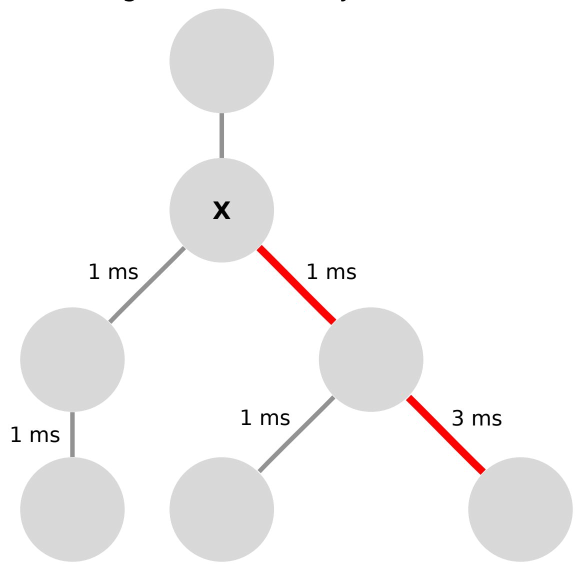
To conclude, it should be noted that this approach constitutes a powerful way to surveil the performance of an actor and, by extension, the performance of the entire execution graph. For each metric (frequency and latency), each actor has one monitor. A monitor contains code that is being executed repeatedly throughout the lifecycle of an actor. The methods embodying this code are invoked with run-time information about the actor. Moreover, monitors are able to communicate with each other using messages. As monitors are nothing but classes extending one of the traits `LeafNodeMonitor`, `UnaryNodeMonitor` or `BinaryNodeMonitor`, they can leverage the capabilities of Scala classes, e.g., keeping state in field variables of the class.

As mentioned before, EventScala comes with one exemplary frequency monitor as well as one exemplary latency monitor. The following two paragraphs will informally describe them.

The exemplary frequency monitor, called average frequency monitor, is comprised of four classes, which are all defined in one file `AverageFrequencyMonitor.scala`. When setting up EventScala's execution engine to use it for frequency monitoring, the user needs to supply an interval, e.g., 30 seconds. The monitor instance of each actor of the resulting execution graph uses a field variable to keep track of the number of events emitted by that actor. Whenever the interval is up, e.g., every 30 seconds, the monitor uses that number to calculate whether `FrequencyRequirements` have been met. Then, it sets the counter back to zero. For example, assume that a query that is represented by a monitor's actor contains the requirement `frequency > ratio(2.instances, 5.seconds) /* ... */`. If the actor emitted 12 events in the last 30 seconds, the monitor would calculate that—on *average*—the actor emits 2 events in 5 seconds. This would not satisfy the requirement, so the monitor would invoke the closure of the requirement.

The exemplary latency monitor, called path latency monitor, is also comprised of four classes, all of which are located in one file called `PathLatencyMonitor.scala`. This is a more complex monitor as it leverages communication between the monitor instances using messages. Periodically, each monitor calculates the so-called path latency of its respective actor. Given all paths that connect the actor in question to `LeafNode`s, path latency denotes the time it takes a message, i.e., an event, to travel along the slowest of these paths. Figure 4 illustrates this concept by highlighting the path that is used to calculate the path latency of a node "X". Obviously, the path latency of a `LeafNode` is always 0. In order to calculate it for `UnaryNode`s and `BinaryNode`s, each monitor surveiling such a node periodically sends a message to its child's or its children's monitor(s), respectively, which will respond immediately. The time that passes between sending the message and receiving the response is being divided by 2 in order to approximate the time it takes for an event to travel from a child actor to the actor in question. This value is referred to as child latency. Obviously, child latency equals path latency if the child actor of the actor in question

Figure 4: Path latency illustration



is a LeafNode. Path latency values are always advertised to the parent actor's monitor. A UnaryNode's monitor can simply add the advertised path latency of its child actor to its child latency in order to obtain its own path latency. A BinaryNode's monitor has to do so twice, i.e., it has to add the advertised path latency of child actor 1 to the child latency of child actor 1, and then do so again for child actor 2. The greater of the two resulting values will be adopted as the BinaryNode's path latency. If a LatencyRequirement has been defined over the query that an actor represents, the requirement's closure will be invoked if the calculated path latency does not satisfy the requirement. Figure 4 illustrates how to calculate path latency: Of the 3 paths that connect actor "X" to a LeafNode, the one indicated by the thick, red edges is the slowest. Therefore, it is used to calculate "X"'s path latency, which is 1 ms + 3 ms = 4 ms. When setting up EventScala's execution engine to use the path latency monitor for latency monitoring, an interval has to be provided, denoting how often path latency values are to be re-calculated, thus, denoting how often latency requirements are to be checked.

5 Simulation

In order to showcase its capabilities, EventScala comes with an exemplary set up simulating a real-world use case. It is comprised of three main components:

- 4 sample `Publishers` representing four streams that can be subscribed to
- 2 sample queries expressed using the DSL
- 1 sample configuration for the execution engine

In this section, all three components are described in detail. Then, the output that is being generated when running the simulation is discussed.

In EventScala, a publisher represents a primitive stream that can be subscribed to. To this end, the specifications of the `Publisher` trait (which, in turn, extends the `Actor` trait) include that a publisher has to maintain a set of subscribers. On top of that, EventScala defines a case class `RandomPublisher`, which publishes one event every x milliseconds, with x being a pseudo-random `Integer` between 0 (inclusive) and 5000 (exclusive). A `RandomPublisher` takes a closure of type `Integer => Event`, which it invokes in order to obtain the `Event` instances it publishes. To obtain the first event, it calls the closure with `0`, to invoke the second event, it calls the closure with 1, and so on. In the simulation, there are four `RandomPublishers` called “A”, “B”, “C” and “D”, representing streams of the types `Stream1[Int]`, `Stream1[Int]`, `Stream1[Float]` and `Stream1[String]`, respectively, as shown in listing 11.

Listing 11: The four `RandomPublishers` of the simulation

```
val publisherA: ActorRef = actorSystem.actorOf(Props(
  RandomPublisher(id => Event1(id))), "A")
val publisherB: ActorRef = actorSystem.actorOf(Props(
  RandomPublisher(id => Event1(id * 2))), "B")
val publisherC: ActorRef = actorSystem.actorOf(Props(
  RandomPublisher(id => Event1(id.toFloat))), "C")
val publisherD: ActorRef = actorSystem.actorOf(Props(
  RandomPublisher(id => Event1(s"String($id)"))), "D")
```

The two sample queries `query1` and `query2` (shown in listing 12) are expressed using the DSL and should not require any further explanation. `query1` will not be presented below as it is the same query that is shown in listing 9.

Listing 12: `query2` of the simulation

```
val query2: Query4[Int, Int, Float, String] =
  stream[Int]("A")
  .and(stream[Int]("B"))
  .join(
    sequence(
      nStream[Float]("C") -> nStream[String]("D"),
      frequency > ratio(1.instances, 5.seconds) otherwise { (nodeData) =>
        println(s"PROBLEM:\tNode ‘${nodeData.name}‘ emits too few events!") }),
    slidingWindow(3.seconds),
    slidingWindow(3.seconds),
    latency < timespan(1.milliseconds) otherwise { (nodeData) =>
      println(s"PROBLEM:\tEvents reach node ‘${nodeData.name}‘ too slowly!") })
```

Lastly, the simulation comes with a sample configuration for the execution graph that EventScala's execution engine creates for either `query1` and `query2`. (Shown in listing 13 is the version for `query1`.) It is specified that the exemplary monitors, i.e., `AverageFrequencyMonitor` and `PathLatencyMonitor`, will be used for frequency and latency monitoring, respectively. The former is set to perform its calculations every 15 seconds while the latter is set to do so every 5 seconds. Both of them are set to log their calculations to the console. The closure that will be invoked when the graph has been created simply prints

"STATUS:\tGraph has been created." to the console, and the closure that will be called whenever the root actor produced an event simply prints said event to the console. It is to be noted that the latter closure is type-safe, i.e., its type is Either[Int, String], Either[Int, X], Either[Float, X] => Any.

Listing 13: The execution graph configuration of the simulation (Newlines were added to fit the page.)

```
val graph: ActorRef = GraphFactory.create(
  actorSystem =               actorSystem,
  query =                     query1,
  publishers =                publishers,
  frequencyMonitorFactory =   AverageFrequencyMonitorFactory
                              (interval = 15, logging = true),
  latencyMonitorFactory =     PathLatencyMonitorFactory
                              (interval =  5, logging = true),
  createdCallback =           () => println("STATUS:\tGraph has been created."))(
  eventCallback =             {
    case (Left(i1), Left(i2), Left(f)) =>
      println(s"COMPLEX EVENT:\tEvent3($i1,$i2,$f)")
    case (Right(s), _, _) =>
      println(s"COMPLEX EVENT:\tEvent1($s)")
    // This is necessary to avoid warnings about non-exhaustive ‘match‘:
    case _ =>
  })
```

When being run for approximately 15 seconds (not including set up time) on a Lenovo ThinkPad X220i laptop with a Intel Core i3-2350M CPU @ 2.30 GHz * 2 processor, this simulation generates the console output like shown in listing 14.

Listing 14: The console output of the simulation (Newlines were added to fit the page.)

```
STATUS:         Graph has been created.
LATENCY:
  Events reach node ‘disjunction-1-conjunction-1-selfjoin-1-dropelem‘ after
  PT0.015S. (Calculated every 5 seconds.)
PROBLEM:
  Events reach node ‘disjunction-1-conjunction-1-selfjoin-1-dropelem‘ too slowly!
STREAM A:       Event1(1)
STREAM B:       Event1(2)
STREAM A:       Event1(2)
STREAM D:       Event1(String(2))
COMPLEX EVENT:  Event1(String(2))
STREAM C:       Event1(1.0)
COMPLEX EVENT:  Event3(2,2,1.0)
STREAM B:       Event1(4)
STREAM A:       Event1(3)
STREAM C:       Event1(2.0)
COMPLEX EVENT:  Event3(4,4,2.0)
LATENCY:
  Events reach node ‘disjunction-1-conjunction-1-selfjoin-1-dropelem‘ after
  PT0.0015S. (Calculated every 5 seconds.)
PROBLEM:
  Events reach node ‘disjunction-1-conjunction-1-selfjoin-1-dropelem‘ too slowly!
STREAM D:       Event1(String(3))
COMPLEX EVENT:  Event1(String(3))
STREAM C:       Event1(3.0)
STREAM B:       Event1(6)
STREAM D:       Event1(String(4))
COMPLEX EVENT:  Event1(String(4))
```

```
STREAM C:       Event1(4.0)
STREAM B:       Event1(8)
STREAM A:       Event1(4)
COMPLEX EVENT:  Event3(8,8,3.0)
STREAM C:       Event1(5.0)
STREAM B:       Event1(10)
COMPLEX EVENT:  Event3(10,10,5.0)
STREAM D:       Event1(String(5))
COMPLEX EVENT:  Event1(String(5))
LATENCY:
  Events reach node ‘disjunction-1-conjunction-1-selfjoin-1-dropelem‘ after
  PT0.0005S. (Calculated every 5 seconds.)
STREAM C:       Event1(6.0)
STREAM A:       Event1(5)
COMPLEX EVENT:  Event3(10,10,6.0)
STREAM B:       Event1(12)
STREAM B:       Event1(14)
STREAM A:       Event1(6)
STREAM C:       Event1(7.0)
COMPLEX EVENT:  Event3(12,12,7.0)
STREAM C:       Event1(8.0)
STREAM D:       Event1(String(6))
COMPLEX EVENT:  Event1(String(6))
LATENCY:
  Events reach node ‘disjunction-1-conjunction-1-selfjoin-1-dropelem‘ after
  PT0.0015S. (Calculated every 5 seconds.)
PROBLEM:
  Events reach node ‘disjunction-1-conjunction-1-selfjoin-1-dropelem‘ too slowly!
FREQUENCY:
  On average, node ‘disjunction-1-conjunction-1-selfjoin‘ emits 3 events every
  5 seconds. (Calculated every 15 seconds.)
PROBLEM:
  Node ‘disjunction-1-conjunction-1-selfjoin‘ emits too few events!
FREQUENCY:
  On average, node ‘disjunction-1-conjunction-1-selfjoin‘ emits 9 events every
  15 seconds. (Calculated every 15 seconds.)
```

It can be observed that publishers print the primitive events they publish to the console. Likewise, the root node prints the complex events the execution graph produces to the console. As expected, the latency monitor of the node `disjunction-1-conjunction-1-selfjoin-1-dropelem` logged to the console four times, i.e., once after graph creation, once after 5 seconds, once after 10 seconds, and once after 15 seconds. Also as expected, the frequency monitor of the node `disjunction-1-conjunction-1-selfjoin` only logged to the console once, i.e., after 15 seconds. Lastly, it is to be noted that the latency requirement has been met in 1 out of 4 times. The two frequency requirements, however, are defined in a way that they can never both be met. Accordingly, after 15 seconds, one of them was met (`frequency < ratio(12.instances, 15.seconds)`) while the other one was not (`frequency > ratio(3.instances, 5.seconds)`).

6 Conclusion

This thesis set out to explore ways to solve some of the existing problems of current EP solutions, i.e., the communication of queries in the form of strings (e.g., [3]), the absence of concurrent execution from industry-grade solutions [4] and the missing ability to express QoS requirements as part of a query. To this end, the EventScala framework has been proposed in this thesis, constituting a type-safe, distributed and QoS-oriented approach to EP.

It can be concluded that using EventScala's DSL to express queries is an approach superior to expressing them in the form of strings. Errors in such strings cannot be detected at compile-time, therefore, they will cause failure at run-time. To the contrary, queries expressed using the DSL are nothing but Scala expressions, thus, they are type-checked at compile-time. Syntactic errors such as misspelled method names are caught during compilation just like type-related errors, e.g., the `dropElem3` operator cannot be applied to a stream of events that consist of only two or less elements.

Furthermore, it can be stated that EventScala introduces a highly distributed approach to executing queries. With each primitive and operator of a query being represented by a processing node, i.e., by an Akka actor, the benefits of concurrent execution can be reaped. For example, parallel execution can be employed for performance gains, parts of the processing can be moved to be in proximity of publishers, etc.

Lastly, an approach to support the annotation of queries with QoS requirements at the language as well as the execution level has been presented. The proposed DSL represents a way to incorporate such requirements on a per-query basis into a query language. EventScala's execution engine constitutues a novel way to run a query while monitoring run-time performance and reacting to non-met requirements.

As the EventScala framework is the contribution of a bachelor thesis, it is merely a prototype, missing many of the features of industry-grade solutions. E.g., at the execution level, fault tolerance has not been considered. Nevertheless, as successfully demonstrated with the simulation, EventScala can be used to define and run non-trivial queries, featuring stream subscriptions from various publishers, operators from both SP and CEP as well as several QoS requirements.

7 References

[1] Annika Hinze, Kai Sachs, and Alejandro Buchmann. “Event-based Applications and Enabling Technologies”. In: *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*. DEBS ’09. Nashville, Tennessee: ACM, 2009, 1:1–1:15. ISBN: 978-1-60558-665-6. DOI: 10.1145/1619258.1619260. URL: http://doi.acm.org/10.1145/1619258.1619260.

[2] Alejandro Buchmann et al. “From Calls to Events: Architecting Future BPM Systems”. In: *Proceedings of the 10th International Conference on Business Process Management*. BPM’12. Tallinn, Estonia: Springer-Verlag, 2012, pp. 17–32. ISBN: 978-3-642-32884-8. DOI: 10.1007/978-3-642-32885-5_2. URL: http://dx.doi.org/10.1007/978-3-642-32885-5_2.

[3] EsperTech Inc. *Esper: Event Processing for Java*. URL: http://www.espertech.com/products/esper.php (visited on 03/16/2017).

[4] Björn Schilling et al. “Distributed Heterogeneous Event Processing: Enhancing Scalability and Interoperability of CEP in an Industrial Context”. In: *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*. DEBS ’10. Cambridge, United Kingdom: ACM, 2010, pp. 150–159. ISBN: 978-1-60558-927-5. DOI: 10.1145/1827418.1827453. URL: http://doi.acm.org/10.1145/1827418.1827453.

[5] Lightbend Inc. *Akka: Build powerful concurrent & distributed applications more easily.* URL: http://akka.io/ (visited on 03/16/2017).

[6] Carl Hewitt, Peter Bishop, and Richard Steiger. “A Universal Modular ACTOR Formalism for Artificial Intelligence”. In: *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*. IJCAI’73. Stanford, USA: Morgan Kaufmann Publishers Inc., 1973, pp. 235–245. URL: http://dl.acm.org/citation.cfm?id=1624775.1624804.

[7] K. Chandy and W. Schulte. *Event Processing: Designing IT Systems for Agile Companies (Special Edition Compliments of IBM)*. 1st ed. New York, NY, USA: McGraw-Hill, Inc., 2009. ISBN: 978-0-07-163711-4.

[8] Opher Etzion and Peter Niblett. *Event Processing in Action*. 1st. Greenwich, CT, USA: Manning Publications Co., 2010. ISBN: 1935182218, 9781935182214.

[9] Sharma Chakravarthy and Qingchun Jiang. *Stream Data Processing: A Quality of Service Perspective Modeling, Scheduling, Load Shedding, and Complex Event Processing*. 1st. Springer Publishing Company, Incorporated, 2009. ISBN: 0387710027, 9780387710020.

[10] Sharma Chakravarthy et al. “Composite Events for Active Databases: Semantics, Contexts and Detection”. In: *Proceedings of the 20th International Conference on Very Large Data Bases*. VLDB ’94. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994, pp. 606–617. ISBN: 1-55860-153-8. URL: http://dl.acm.org/citation.cfm?id=645920.672994.

[11] The Apache Software Foundation. *Apache Flink: Scalable Stream and Batch Data Procesing*. URL: https://flink.apache.org/ (visited on 03/16/2017).

[12] The Apache Software Foundation. *FlinkCEP - Complex event processing for Flink*. URL: https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/libs/cep.html (visited on 03/16/2017).

[13] Sharma Chakravarthy and Raman Adaikkalavan. “Events and Streams: Harnessing and Unleashing Their Synergy!” In: *Proceedings of the Second International Conference on Distributed Event-based Systems*. DEBS ’08. Rome, Italy: ACM, 2008, pp. 1–12. ISBN: 978-1-60558-090-6. DOI: 10.1145/1385989.1385991. URL: http://doi.acm.org/10.1145/1385989.1385991.

[14] U. Dayal et al. “The HiPAC Project: Combining Active Databases and Timing Constraints”. In: *SIGMOD Rec.* 17.1 (Mar. 1988), pp. 51–70. ISSN: 0163-5808. DOI: 10.1145/44203.44208. URL: http://doi.acm.org/10.1145/44203.44208.

[15] Stella Gatziu, Hans Fritschi, and Anca Vaduva. *SAMOS an Active Object-Oriented Database System: Manual*. Tech. rep. 1996.

[16] Raman Adaikkalavan and Sharma Chakravarthy. “SnoopIB: Interval-based Event Specification and Detection for Active Databases”. In: *Data Knowl. Eng.* 59.1 (Oct. 2006), pp. 139–165. ISSN: 0169-023X. DOI: 10.1016/j.datak.2005.07.009. URL: http://dx.doi.org/10.1016/j.datak.2005.07.009.

[17] Daan Leijen and Erik Meijer. “Domain Specific Embedded Compilers”. In: *Proceedings of the 2Nd Conference on Domain-specific Languages*. DSL ’99. Austin, Texas, USA: ACM, 1999, pp. 109–122. ISBN: 1-58113-255-7. DOI: 10.1145/331960.331977. URL: http://doi.acm.org/10.1145/331960.331977.

[18] Jevgeni Kabanov and Rein Raudjärv. “Embedded Typesafe Domain Specific Languages for Java”. In: *Proceedings of the 6th International Symposium on Principles and Practice of Programming in Java*. PPPJ ’08. Modena, Italy: ACM, 2008, pp. 189–197. ISBN: 978-1-60558-223-8. DOI: 10.1145/1411732.1411758. URL: http://doi.acm.org/10.1145/1411732.1411758.

[19] Daniel Spiewak and Tian Zhao. “ScalaQL: Language-integrated Database Queries for Scala”. In: *Proceedings of the Second International Conference on Software Language Engineering*. SLE’09. Denver, CO: Springer-Verlag, 2010, pp. 154–163. ISBN: 3-642-12106-3, 978-3-642-12106-7. DOI: 10.1007/978-3-642-12107-4_12. URL: http://dx.doi.org/10.1007/978-3-642-12107-4_12.

[20] Debasish Ghosh. *DSLs in Action*. 1st. Greenwich, CT, USA: Manning Publications Co., 2010. ISBN: 9781935182450.

[21] Lightbend Inc. *Slick: Functional Relational Mapping for Scala*. URL: http://slick.lightbend.com/ (visited on 03/16/2017).

[22] C. Liebig, M. Cilia, and A. Buchmann. “Event composition in time-dependent distributed systems”. In: *Proceedings Fourth IFCIS International Conference on Cooperative Information Systems. CoopIS 99 (Cat. No.PR00384)*. 1999, pp. 70–78. DOI: 10.1109/COOPIS.1999.792159.

[23] Peter R. Pietzuch, Brian Shand, and Jean Bacon. “A Framework for Event Composition in Distributed Systems”. In: *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*. Middleware ’03. Rio de Janeiro, Brazil: Springer-Verlag New York, Inc., 2003, pp. 62–82. ISBN: 3-540-40317-5. URL: http://dl.acm.org/citation.cfm?id=1515915.1515921.

[24] A. Zeidler and L. Fiege. “Mobility support with REBECA”. In: *23rd International Conference on Distributed Computing Systems Workshops, 2003. Proceedings.* May 2003, pp. 354–360. DOI: 10.1109/ICDCSW.2003.1203579.

[25] A. Biger, Y. Rabinovich, and O. Etzion. “Stratified Implementation of Event Processing Network”. In: *Fast abstract in DEBS, 2008*. 2008.

[26] Amer Farroukh et al. “Parallel Event Processing for Content-based Publish/Subscribe Systems”. In: *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*. DEBS ’09. Nashville, Tennessee: ACM, 2009, 8:1–8:4. ISBN: 978-1-60558-665-6. DOI: 10.1145/1619258.1619269. URL: http://doi.acm.org/10.1145/1619258.1619269.

[27] Rohit Khandekar et al. “COLA: Optimizing Stream Processing Applications via Graph Partitioning”. In: *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*. Middleware ’09. Urbanna, Illinois: Springer-Verlag New York, Inc., 2009, 16:1–16:20. URL: http://dl.acm.org/citation.cfm?id=1656980.1657002.

[28] Martin Hirzel. “Partition and Compose: Parallel Complex Event Processing”. In: *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*. DEBS ’12. Berlin, Germany: ACM, 2012, pp. 191–200. ISBN: 978-1-4503-1315-5. DOI: 10.1145/2335484.2335506. URL: http://doi.acm.org/10.1145/2335484.2335506.

[29] Beate Ottenwälder et al. “MigCEP: Operator Migration for Mobility Driven Distributed Complex Event Processing”. In: *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*. DEBS ’13. Arlington, Texas, USA: ACM, 2013, pp. 183–194. ISBN: 978-1-4503-1758-0. DOI: 10.1145/2488222.2488265. URL: http://doi.acm.org/10.1145/2488222.2488265.

[30] Stefan Appel, Kai Sachs, and Alejandro Buchmann. “Quality of Service in Event-based Systems”. In: GvD Workshop. Bad Helmstedt, Germany, 2010.

[31] S. Behnel, L. Fiege, and G. Muhl. “On Quality-of-Service and Publish-Subscribe”. In: *26th IEEE International Conference on Distributed Computing Systems Workshops (ICDCSW’06)*. July 2006, pp. 20–20. DOI: 10.1109/ICDCSW.2006.77.

[32] Miles Sabin and contributors. *Shapeless: Generic programming for Scala*. URL: https://github.com/milessabin/shapeless/ (visited on 03/16/2017).

[33] Martin Odersky. *Pimp my Library*. URL: http://www.artima.com/weblogs/viewpost.jsp?thread=179766 (visited on 03/16/2017).

[34] JetBrains s.r.o. *Intellij IDEA: The Java IDE*. URL: https://www.jetbrains.com/idea/ (visited on 03/16/2017).

[35] Lightbend Inc. *Akka Scala Documentation*. URL: http://doc.akka.io/docs/akka/2.4/AkkaScala.pdf (visited on 03/16/2017).

[36] Frank Sauer. *CEP with Akka and Esper or Streams*. URL: http://www.lightbend.com/activator/template/akka-with-esper (visited on 03/16/2017).

[37] Facebook Inc. *React: A JavaScript library for building user interfaces*. URL: https://facebook.github.io/react/ (visited on 03/16/2017).

[38] Facebook Inc. *React.Component*. URL: https://facebook.github.io/react/docs/react-component.html (visited on 03/16/2017).