Complex Event Processing: Language-level Integration into Scala

Complex Event Processing: Integration in Scala auf Sprachebene
Bachelor-Thesis von Mark Goldenstein
November 2013

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik
Software Technology Group

Complex Event Processing: Language-level Integration into Scala
Complex Event Processing: Integration in Scala auf Sprachebene

Vorgelegte Bachelor-Thesis von Mark Goldenstein

1. Gutachten: Prof. Dr.-Ing. Mira Mezini
2. Gutachten: Dr. Guido Salvaneschi

Tag der Einreichung:

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, den November 11, 2013

(Mark Goldenstein)

Abstract

Some programming languages provide language-level integration for event processing. Object-oriented languages offer language constructs for imperatively triggered events as object attributes. Aspect-oriented languages feature implicit events that are defined as identifiable points in the control flow of a program. However, such a language-level integration is not yet available for *complex event processing* (CEP) where event streams from multiple sources are combined to infer higher-level and more abstract complex events. In this thesis we propose an approach to introduce language-level support for CEP into Scala.

Zusammenfassung

Manche Programmiersprachen haben eine tiefe Integration für die Verarbeitung von Ereignissen: Objektorientierte Sprachen enthalten Sprachkonstrukte, die es ermöglichen Ereignisse als explizit auslösbare Objektattribute auszudrücken. Aspektorientierte Sprachen unterstützen implizite Ereignisse, die durch identifizierbare Punkte im Ablauf eines Programms definiert sind. Diese Art von Sprachintegration ist jedoch noch nicht für *Complex Event Processing* (CEP) verfügbar, eine Technik bei der Ereignisse aus einer Vielzahl von Quellen miteinander kombiniert werden um abstraktere komplexe Ereignisse abzuleiten. In dieser Thesis besprechen wir einen Ansatz zur Integration von CEP in Scala auf Sprachebene.

Contents

1 Introduction

Most contemporary computer programs make use of event-processing. Event-driven programming is used for the processing of real-time data, reactive behaviors of user interfaces, control automation, and more. Examples of events in different domains are, in a computer application, the click on a user interface element, at the stock market, the update *tick* of a security, and, on a news website, the publication of a new article [1].

Noticeably, events have found their way into programming languages: there are languages (e.g. C#) that implement events natively as language constructs. However, most languages (e.g. Java) do not explicitly implement them but instead rely on design patterns like the Observer pattern.

In Scala, events are handled similarly to Java. However, EScala [2], a DSL and language extension to Scala, defines events as object fields (similarly to C#) and introduces an event expression language which allows developers to *compose* and *transform* existing events into new events.

To understand the meaning of events and complex events, and to classify the different approaches in this context, we will first look into their definitions.

Event Processing is a method of tracking and analyzing (processing) streams of information (data) about things that happen (events) and deriving conclusions from them. *Complex Event Processing* (CEP) is event processing that combines multiple sources of simple events or event patterns to higher-level and more abstract complex events. The goal of CEP is to identify meaningful events (such as opportunities or threats) and respond to them appropriately and as quickly as possible [3].

While both C# and EScala support simple events in an intuitive object-oriented programming style, this breed of events offers only limited functionality in the context of *Big Data*, the emergent trend in modern computing. Today data sources like sensors (RFID[1], NFC[2]), web activities, transactions, social networks, etc. create an avalanche of data that requires efficient and fast processing for which traditional DBMS are not applicable because they cannot fulfill the requirements of timeliness coming from such domains [4]. Here we need new methods, like CEP, which has proven useful in processing Big Data [5].

CEP is widely used in algorithmic trading, surveillance and alarm systems, network monitoring, intrusion detection, traffic management and many other areas [1], [4].

Often, none of the individual events stand out when considered on their own but the combination of events in a short time frame can be significant. When isolated events, sometimes hundreds or thousands, are aggregated and correlated to higher-level and more abstract complex events, a coherent picture of the situation can be built which has meaning at the business rather than on the transactional level. It is this ability to automatically detect and react to subtle cues that makes CEP so powerful [6].

Available CEP engines from both academia and industry include Cayuga [7], Aurora [8], Borealis [9], Esper [10], StreamBase, and Oracle CEP 10g, just to name a few. They use an SQL-like query language and do not provide a deep level of integration with programming languages and object-oriented programming paradigms.

[1] Radio-frequency identification

[2] Near field communication

In summary, we have noted the existence of (*i*) programming languages with language-level support for events and (*ii*) programming libraries with support for complex event processing but without a deep language-level integration. There is a missing spot, namely a programming language or library that supports CEP and, at the same time, provides straight-forward language-level integration. This spot is the focus of our research.

Our thesis provides the following contribution:

- We provide CEScala, a DSL embedded in Scala that features the expressivity of CEP libraries without relinquishing the level of language integration of events offered by event-based languages.
- We provide a type-safe implementation of CEScala that incorporates the same API as EScala.
- We evaluate the performance of our implementation and highlight the design factors that influence efficiency.

The thesis is structured as follows. In section 2 we overview the related work. In section 3 we present the design of CEScala. In section 4 we discuss the significant details of the implementation. We evaluate our work in section 5. In section 6 we conclude the thesis and outline future work.

2 State of the Art

This section discusses related work. The design of CEScala combines elements from event-driven programming, CEP libraries, and Language-integrated Query implementations. We discuss the relation of our work to each of these areas in dedicated subsections.

2.1 Event Processing and Event-based Languages

Aspect-oriented programming (AOP), an example implementation is AspectJ [11], is a technique that improves separation of concerns in software. AOP provides language mechanisms that explicitly capture crosscutting structure. It features *implicit events* aka join points: events that are not explicitly declared but are instead defined as identifiable points in the control-flow of a program such as the end of a method invocation. AOP avoids the explicit triggering of events at these points by letting *aspects* observe them directly. This simplifies the code and reduces the need for preplanning.

EScala [2] is a DSL that implements C#-like events in Scala. It is a language design that combines imperatively triggered events with AOP mechanisms like implicit events. At its core EScala facilitates stitching together different components using events, which avoids the verbose boilerplate code that is needed when using the Observer pattern instead. Events are represented as object attributes and have to be referenced in this way to trigger them or attach new reactions. At the same time, EScala introduces an event expression language that enables composing and transforming events, cf. section 3.1.3.

Ptolemy [12] is a language that combines ideas of imperative events and AOP. Unlike EScala, Ptolemy does not provide implicit events. Event handlers react to events characterized by their type which fully decouples event sources and sinks.

An alternative to events as object attributes are publish-subscribe systems [13] which achieve a higher degree of decoupling by introducing a global event system in which some components publish events and other components subscribe to them. Thus, such systems have no direct connection between objects that trigger events and objects that react to them. However, this high-level degree of decoupling is not always desirable because it makes the software more difficult to understand and maintain [2].

2.2 Complex Event Processing Systems

An increasing number of distributed applications require processing continuously flowing data at an unpredictable rate to obtain timely responses to complex queries. Traditional DBMSs require data to be (persistently) stored and indexed before it can be processed, and the processing only occurs when explicitly asked by the user, i.e. asynchronously with respect to its arrival. Therefore, storing events in a traditional database and then querying it yields very poor performance. Both aspects contrast with the requirements of real-time applications which include processing data as soon as it becomes available and discarding irrelevant data. These requirements have led to the development of systems specifically designed to process information as a flow, according to a set of pre-deployed processing rules. Despite having a common goal, these systems differ in a wide range of aspects, including architecture, data models, rule languages, and processing mechanisms. Two models have emerged and are today competing: the *data stream processing* model and the *complex event processing* model [4].

Data Stream Processing systems use query languages that have very limited expression power and only allow simple selection predicates. They trade expressiveness for performance [14]. Examples of such systems are Aurora [8] and Borealis [9]. When well engineered, they exhibit very high scalability in both the number of queries and the stream rate. However, their inability to express queries that span multiple input events makes them unsuitable for complex event processing [7].

Esper [10] and Cayuga [7] feature full CEP support. Since Cayuga is designed to leverage the traditional publication-subscription techniques, it allows high scalability, and its system architecture also supports a large number of concurrent subscriptions [14].

Esper is similar to Cayuga but it also has the ability to express complex matching conditions that includes temporal windows, joining of different event streams, as well as filtering and sorting them. Furthermore, it also has the ability to detect sequences and patterns of unrelated events.

There are also several commercial CEP engines in the market but most of them have been built on top of open source CEP engines such as StreamBase on Aurora and Oracle CEP 10g on Esper [15]. Details about them have limited availability due to their commercial nature.

2.3 Language-integrated Query

Because most CEP engines are controlled using an SQL-like language, much can be learned from examining how relational databases are accessed in modern programming languages.

One of the primary aspects of this situation is impedance mismatch between the relational model and the paradigm employed by most general-purpose programming languages. Concepts are expressed very differently in a relational database than in a standard memory model. The prevalent solution to this problem of conceptual orthogonality is to give up attempting to adapt one world to the other and let the different conceptual paradigms remain separate. In this solution the application layer retrieves data as necessary from the relational store by using concepts native to a relational database: declarative query languages such as SQL. This is by far the simplest approach to application-level database access but it comes with significant disadvantages [16].

Unlike the situation for “normal” programming language constructs, a compiler is not aware of the semantics of embedded database queries, and thus offers no help regarding their well-formedness checking or their processing. Approaches to overcome these shortcomings fall under the general umbrella of *language-integrated query*, of which *embedded SQL* is an early example and Microsoft LINQ today’s best-known representative [17].

Implementations of language-integrated query in Scala include Squeryl [18], ScalaQL [16], ScalaQuery, and Slick which is part of the Typesafe Stack. Those implementations define their own DSLs for accessing relational databases. Internally, they create representations of the desired queries in *abstract syntax trees* which enables the Scala compiler to check queries for well-formedness and sometimes even optimize them.

3 The CEScala Library

This section presents CEScala, a Scala library with support for complex event-driven designs based on EScala [2]. In a nutshell, CEScala combines the interface of EScala with Esper, a CEP engine written in Java. Because EScala is fully integrated into the Scala language, it is better suited for object-oriented programming than Esper's cumbersome own SQL-like API, cf. section 2.3. Thus, CEScala merges the advantages of the expressivity and the complex event processing capabilities of Esper with the simplicity and language-level integration of the EScala interface.

3.1 The EScala DSL in a nutshell

Because CEScala is based on EScala's interface, the syntax that is used for event declaration, composition and reaction binding corresponds to that of EScala. We describe this syntax in the following subsections.

3.1.1 Event Declaration

Events can be either *imperative* or *declarative*. They can collect data about the context of their occurrence. The type of the data has to be specified using Scala's type system by adding the type in square brackets after the event. If an event should not collect any data, the `Unit` type has to be used. By convention, events collecting several types of data can be specified using tuple types. Custom types are supported but they have to be declared at the top-level (i.e. they cannot be nested).[1] We recommend making events and event properties immutable (in accordance to [10]), however this is not a hard requirement and CEScala also accepts events that are mutable.

Imperative events are declared using the `ImperativeEvent` class and must be explicitly triggered. *Declarative events* are defined by *event expressions* which compose and transform other *event expressions*, including *imperative events*, cf. section 3.1.3.

3.1.2 Binding Reactions to Events

Analogous to C#, reactions can be registered with events by expressions of the form `ev += react` where `ev` denotes the event name and `react` denotes a function implementing the reaction. Similarly, reactions can be unregistered by expressions of the form `ev -= react`. A reaction to an event can be any function whose type is compatible with the data type of the event, i.e. the argument type of the function must be equal to the data type of the event or a supertype thereof.

[1] Otherwise Scala's `ClassTag` and `TypeTag` constructs, which are used in the implementation of CEScala, do not work.

3.1.3 Event Expression Language

Events are syntactically defined by *event expressions*. The simplest event expressions are already defined events. *Event operators* can then be used to compose and transform other event expressions to define new events. The semantics of event expressions are explained below where `ev1` and `ev2` denote event expressions. Those event expressions are already supported by EScala but we have re-implemented them in CEScala using Esper as the backend.

The expression `ev1 || ev2` denotes the union of `ev1` and `ev2`. This expression matches event occurrences matched by `ev1` or `ev2`. The data type of the expression is the least common supertype of the data types `ev1` and `ev2`.

The expression `ev1 && func` filters `ev1` by an arbitrary boolean function `func` taking the data of the event as a parameter.

The expression `ev1.map(func)` matches an occurrence matched by `ev1` in which case the data collected by `ev1` is transformed by `func`.

3.2 Extending EScala—The CEScala DSL

By using Esper as our event processing engine we can provide several new language constructs.

3.2.1 join

The most significant addition to EScala is event joining: Two different events can be joined together on a specified condition using the `join` operator (quite like the `join` operator in SQL).

When performing a join in CEScala the following parameters must be specified: a reference to the second event, the time or length window of each event, and the join condition. The window argument specifies the time window (e.g. the last minute) or the length window (e.g. the last 100 events) of events in which the join condition is evaluated. The join condition is a predicate that is used to test combinations of the event occurrences. When the left event has n parameters and the right event has m parameters, the joined event is a tuple merging both event types (and is therefore of size $n+m$). The parameters of the left-side event can be accessed via the first n elements of the tuple, the parameters of the right-side event via the last m elements.

The event window is defined using the syntax `time(t)` where `t` denotes the time interval into the past to observe for a time window or `length(l)` where `l` denotes the number of past event to observe for a length window. The time interval `t` is defined using the syntax `n units` where `n` is an integer and `units` is one of the following: `msec`, `sec`, `min`, `hour`, `hours`, `day`, or `days`.

When defining the join condition, event properties are referenced in the same way as tuples in Scala, e.g. `ev1._1` references the first property of the event `ev1`. Condition expressions are defined using the syntax `p1 OP p2` where `p1` and `p2` denote event properties, and `OP` denotes the condition operator. CEScala supports the following condition operators:

- === which returns true when the left event property is equal to the right event property,
- !== which returns true when the left event property is not equal to the right event property,

Listing 3.1: Two events joined on a condition

```
1  var joinedString = ""
2  val ev1 = new ImperativeEvent[Int]
3  val ev2 = new ImperativeEvent[(Int, String)]
4  val ev3 = ev1 join ev2 window time(30 sec) on ((ev1, ev2) => ev1._1 ===
    ev2._1)
5  val r1 = (e: (Int, Int, String)) => testString += e._3
6  ev3 += r1
```

- < which returns true when the left event property is smaller than the right event property,
- > which returns true when the left event property is larger than the right event property,
- <= which returns true when the left event property is smaller than or equal to the right event property, and
- >= which returns true when the left event property is larger than or equal to the right event property.

Condition expressions can be chained using the `&&` and `||` operators for conjunction and disjunction, respectively.

The `join` operator in CEScala comes in two flavours. There is a full syntax and a short one which can be used when both event windows are the same.

The full join expression is `ev1.window(ev1-window) join ev2.window(ev2-window) on ((ev1, ev2) => condition)` where `ev1` and `ev2` denote event expressions, `ev1-window` and `ev2-window` denote the corresponding event windows, and `condition` denotes the join condition.

The short join expression is `ev1 join ev2 window ev-window on ((ev1, ev2) => condition)` where `ev1` and `ev2` denote event expressions, `ev-window` denotes the shared event window, and `condition` denotes the join condition.

For example, Listing 3.1 demonstrates a join using the short syntax. The join includes the events `ev1` and `ev2` (lines 2–3) that were triggered within the last 30 seconds, on the condition that their respective first parameters are equal (line 4). Every time the join is performed, CEScala runs the reaction that is attached to it. In this case, the second parameter of `ev2` (a `String`) is appended to the variable `joinedString` (line 5).

3.2.2 repeat

Sometimes it makes sense to wait for an event to trigger several times before a reaction is executed. This is precisely how the `repeat` operator in CEScala is used. A given number of event occurrences (n) are aggregated in a `Seq` and reactions use this `Seq` as input. After the reaction has been run the aggregation process restarts and when another n events have been collected the reaction is run again on the new batch of events.

The syntax of the repeat event pattern is as follows: `ev repeat n` where `ev` denotes the event name and `n` denotes the number of occurrences to be aggregated. Repeat event expressions cannot be referenced in other event expressions.

Listing 3.2: A repeat event expression

```
val ev1 = new ImperativeEvent[Int]
val ev2 = ev1 repeat 3
val r1 = (e: Seq[Int]) => {
  println("First event: " + e(0))
  println("Second event: " + e(1))
  println("Third event: " + e(2))
}
ev3 += r1
```

Listing 3.2 demonstrates a repeat event expression in which a reaction is run after three event occurrences have been collected. The reaction prints out the values of all collected event occurrences.

4 Implementation

Since the interfaces of CEScala and Esper are radically different, a strategy of mapping CEScala events to Esper must be used. There is a number of design parameters involved, including: How should CEScala's imperative events be mapped to Esper's event streams? Which Esper event representation should be used? Should event listeners be bound to Esper statements as `UpdateListener`s or as `subscriber`s? How should event transformations be handled? This section describes the design choices made during the creation of CEScala and the reasoning behind them.

4.1 Event Representation

To enable compatibility with EScala, CEScala events are defined by value types for events with only one property and by tuple types for events with multiple properties. Since neither is supported by Esper directly, the value or tuple must be converted to an appropriate representation.

Esper supports four different representations of events: POJO, `Map`, `Object[]`, and XML. We have chosen to use Object-array representation because it exhibits the best performance and memory usage characteristics [10].

4.1.1 Mapping Events to Esper

There are two different methods of converting CEScala events to Esper events: The first option is to map each CEScala event declaration to its own event stream in Esper, which is the most obvious way. In contrast, the second option is to group all event declarations with the same property types to merged event streams and then differentiate the CEScala events with an additional event id parameter (which can be filtered by Esper), cf. Figures 4.1 and 4.2. Since complex event processing engines are usually optimized for a high event throughput but not necessarily for a high number of different event streams, the second method might be performance-wise better. However, this depends largely on the internal implementation of complex event processing in Esper. We have opted to implement the basic functionality using both methods in CEScala in order to evaluate them with different benchmarks and then choose the best method in terms of performance, cf. section 5.2. We describe the specifics of the implementation with merged event streams in the following section.

4.1.2 Merged Event Streams

When mapping several CEScala events of the same type to a single event stream in Esper, we need to add an additional event property in Esper which is used to identify the original events. We illustrate this in Figure 4.2.

Listing 4.1 presents the code changes necessary for merged event streams. We create two maps, `propertiesToEsperEventMap` which tracks different property type combinations and maps them to Esper event names, and `cescalaToEsperEventMap` which tracks CEScala event names and maps them to Esper event names. We use the former to decide

Figure 4.1: Separate Event Streams

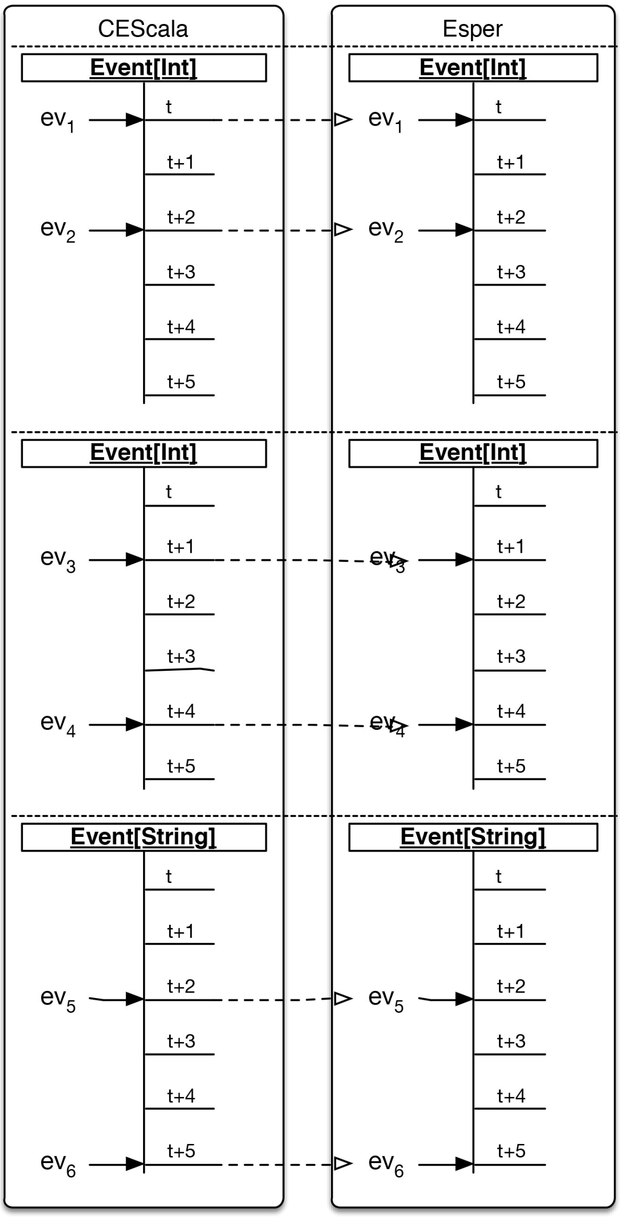
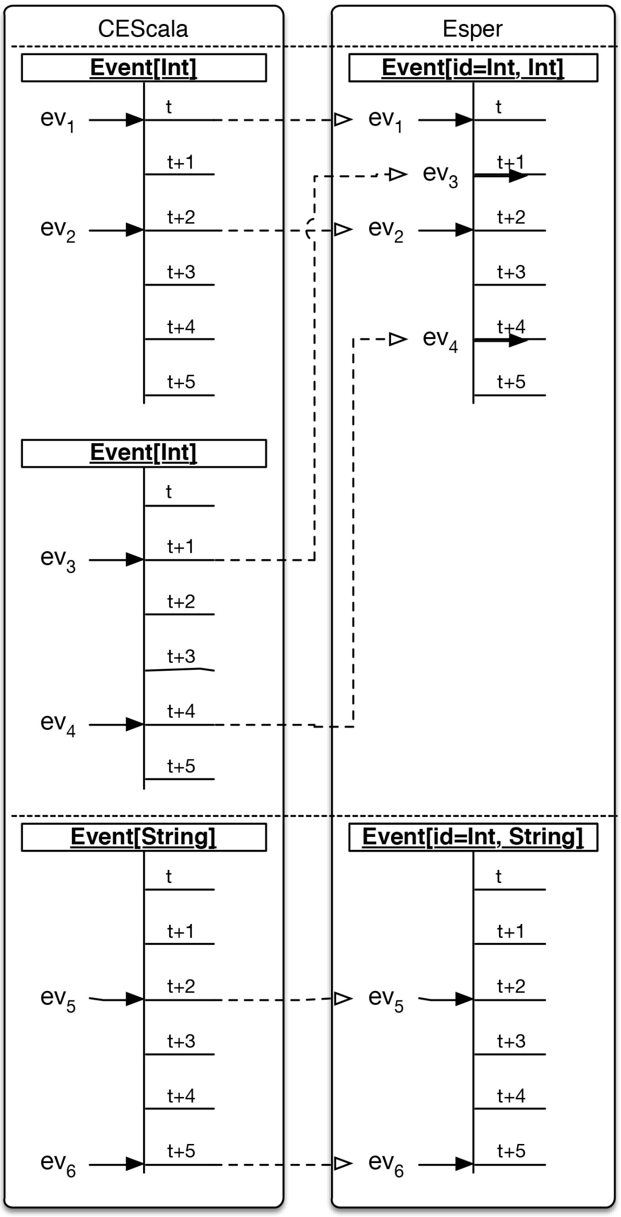


Figure 4.2: Merged Event Streams



Listing 4.1: Merged event streams

```
1  // track different property type combinations
2  val propertiesToEsperEventMap = mutable.HashMap[Array[AnyRef], String]()
3
4  // track which CEScala event corresponds to which Esper event
5  val cescalaToEsperEventMap = mutable.HashMap[String, String]()
6
7  def addEventType(name: String, propertyTypes: Array[AnyRef]) {
8    propertiesToEsperEventMap.get(propertyTypes) match {
9      case Some(_) => // do nothing
10     case _ => [...] // create new event stream
11   }
12 }
13
14 def sendEvent[T: TypeTag](name: String, properties: T) {
15   [...]
16   epService.getEPRuntime.sendEvent([...], cescalaToEsperEventMap.get(
      name).get)
17 }
```

whether we need to create a new Esper stream (lines 8–11) and we use the latter to deduce which Esper event we have to trigger when a CEScala event is triggered (line 16).

We present our performance evaluation of this alternative solution in section 5.2.

4.1.3 Handling Events with Multiple Properties

Another issue is how to handle events with multiple properties. Since the original EScala library endorses using tuples for the propagation of multiple properties, we have chosen to implement CEScala similarly. Because Esper requires registering all event types with property names and classes, it is necessary to differentiate between events with only one property and events with multiple properties (tuples) and also implement a way of identifying the types of properties and tuple elements. We have accomplished this using the Scala Reflection API, more specifically with the help of the recently introduced `ClassTag` and `TypeTag` constructs.

As soon as a new event is declared in CEScala, a correspondig event stream is created in Esper. However, Esper's API requires providing the classes of the event parameters but Scala's types are erased at compile-time (as with other JVM languages). To access type information at runtime, `ClassTags` and `TypeTags` which carry the type information have to be explicitly defined using context bounds, cf. line 1 of Listing 4.2. If the event parameter is a subtype of of `Product` (i.e. it is a tuple), the classes of the parameter types are extracted from the tuple via the `TypeTag` (lines 4–8). Otherwise, if the event parameter is not a subtype of `Product`, it is assumed that it is a simple value, and its class is extracted via the `ClassTag` (line 11).

4.2 Reaction Binding

In contrast to EScala, it is not possible in Esper to bind reactions to event streams directly. Instead, reactions have to be bound to `EPStatements`, standing *Event Processing Language*

Listing 4.2: Using the Scala Reflection API to identify event parameter types

```scala
ImperativeEvent[T: ClassTag : TypeTag] extends EventNode[T] {
  typeOf[T] match {
    case t if t <:< typeOf[Product] => // Type is a tuple
      val typeArgs = t match {
        case TypeRef(_, _, args) => args
      }
      val m = runtimeMirror(getClass.getClassLoader)
      val propertyTypes: Array[AnyRef] = typeArgs.map(t => m.
       runtimeClass(t.typeSymbol.asClass)).toArray
      CEPEngine.addEventType(name, propertyTypes)
    case _ => // Type is not a tuple
      val propertyTypes: Array[AnyRef] = Array(classTag[T].runtimeClass)
      CEPEngine.addEventType(name, propertyTypes)
  }
  [...]
}
```

(EPL) queries which operate on event streams. Therefore, to bind a reaction to an event as it is done in `EScala`, CEScala creates an `EPStatement` which selects all occurrences of an event from the corresponding event stream, and then binds the reaction to this statement.

There are two different methods of binding reactions to `EPStatements` that are supported by Esper: Reactions have to be wrapped either in an `UpdateListener` or in a `subscriber` object. Although the Esper Reference favors using `subscriber` objects for better performance, there is a remarkable disadvantage: Contrary to `UpdateListeners`, `subscriber` objects must provide update methods with signatures that match the number and the types of the properties of the underlying event [10]. Creating `subscriber` objects with the proper signatures in CEScala would require implementing interfaces at runtime.[1]

Furthermore, our performance tests show that using a `subscriber` object does not actually improve the overall performance, cf. section 5.2. Hence, we have decided to go with the simple and concise way, i.e. wrapping reactions in `UpdateListeners`. We present this implementation in Listing 4.3. Esper properties are accessed by their name (which CEScala sets according to the order of their declaration), cf. lines 7 and 9. Properties of events with more than one property have to be converted to tuples before they can be forwarded to the reaction, cf. line 10.

4.3 Transforming events

When events are transformed with the `map` operator, CEScala creates a reaction which performs the transformation, attaches it to the `EPStatement` that selects all occurrences of the original event, and inserts the transformed event into a new event stream. Then a new `EPStatement` is created and all reactions to the mapped event are wrapped in `UpdateListeners` and bound to the new statement.

[1] An implementation could be based on cglib [19], a code generation library for Java.

Listing 4.3: Wrapping reactions in `UpdateListeners`

```
def MyListener(react: T => Unit) = new UpdateListener {
  override def update(newEvents: Array[EventBean], oldEvents: Array[EventBean]) {
    val event = newEvents(0)
    event.getEventType.getPropertyNames match {
      case p if p.length == 0 => react(Unit.asInstanceOf[T])
      case p if p.length == 1 =>
        react(event.get("P1").asInstanceOf[T])
      case p =>
        val properties = for (propertyName <- p) yield event.get(propertyName)
        react(CEPEngine.toTuple(properties).asInstanceOf[T])
    }
  }
}
```

Listing 4.4: Composing events

```
1  CEPEngine.createEPL("insert istream into " + name + " select * from " + ev1.name)
2  CEPEngine.createEPL("insert istream into " + name + " select * from " + ev2.name)
3
4  val statement = CEPEngine.createEPL("select istream * from " + name)
```

4.4 Composing events

When events are composed with the `||` operator, CEScala infers the least common supertype of the composed events and creates a corresponding event stream in Esper. Then CEScala inserts both events into the new (merged) event stream, creates a new `EPStatement` and all reactions to the composed event are wrapped in `UpdateListeners` and bound to the new statement.

In Listing 4.4 we show how we insert both event streams `ev1` and `ev2` into a new event stream `name` using Esper's query language (lines 1-2) and then create an `EPStatement` selecting all occurrences of the new event (line 4).

4.5 Joining events

When events are joined with the `join` operator, CEScala uses Esper to join the corresponding event streams. To construct the condition `String` which is passed to Esper we implicitly convert an `Event` to a `ValueEventExpr` or a `TupleEventExpr`, depending on the type of the event. Those classes then provide methods to denote the corresponding event properties and create boolean expressions (represented by instances of the class `BoolExpr`). Afterwards, those expressions are converted to `String` representations and passed on to Esper.

The properties of both events are inserted into the new (joined) event stream in Esper. Then a new `EPStatement` is created and all reactions to the joined event are wrapped in `UpdateListeners` that combine the properties of both events to a tuple of the combined length. Afterwards, the `UpdateListeners` are attached to the new statement. To abstract over tuple arities and statically support tuple manipulation we use shapeless [20]—a "type class and dependent type based generic programming library for Scala".

4.6 Using the Repeat Event Pattern

Another concept of Esper are event patterns. Event patterns match when an event or multiple events occur that match the pattern's definition. Patterns can also be time-based [10]. As an example of such patterns we have implemented the `repeat` operator (which is translated to `repeat every` in Esper).

The Repeat Event in CEScala creates a new `EPStatement` of the form `select istream e[0], e[1], e[2] from pattern[every [3] e=eventname]` for a `ev1 repeat 3` event expression where `ev1` denotes the event on which the pattern is applied, `e[i]` denotes the (i-1)th occurrence of the event, and `eventname` denotes the internal name of the event stream represented by `ev1`.

The reactions to the Repeat Event are wrapped in `UpdateListeners` which receive the `Array` of events from Esper and transform it to a `Seq` of event properties which can be parsed by the reactions. Then the `UpdateListeners` are attached to the above statement.

5 Evaluation

While evaluating event processing libraries two different metrics can be analyzed. One is the feature-completeness of the library ("Which types of operations does it support?"), the other one is its performance ("How much time does it take to trigger one million events?"). In this section, we look at both of these metrics and thus provide an assessment of CEScala.

5.1 Feature Characteristics

In CEScala we have re-implemented the features of EScala using Esper and also added new event expressions using a subset of Esper's functionality. Those additional expressions include the `join` and the `repeat` operators. The implementation is designed to be extended, therefore it is simple to add new operators.

Since Esper relies heavily on its SQL-like *event processing language*, mapping it to an object-oriented interface proves to be rather difficult. Similar approaches in Java for SQL (Hibernate and JPA) were designed to provide the maximum functionality and expressiveness of querying databases within the constraints of the Java language—they do not provide the ability to use a single programming language and an object-oriented interface to access the underlying engine's features, as CEScala does. Instead, they escape the language with a String-based query language where their imperative APIs are not sufficient [18].

Similar to other language-integrated query implementations (cf. section 2.3), CEScala provides an object-oriented interface and allows the developer to use a single programming language. This enables the Scala compiler to check queries for well-formedness and thus leads to more intuitive programming, better integration in IDEs, and an improved readability of code.

Although the interface of CEScala is based on EScala, they do not always behave exactly the same. Particularly, the `||` operator behaves differently: When both events match an occurrence, the occurrence is selected only once in the composed event using EScala. Using CEScala however, both event occurrences are selected. The reason for these different behaviours lies in the way event compositions are handled by CEScala: Contrary to EScala, CEScala does does not keep track of event composition or transformation. Hence, when two different events streams originate from a single stream and are composed again later on, CEScala does not know about their shared origin and therefore inserts both events into the new stream. Listing 5.1 shows an example of this difference.

Another problem arises when events have too many properties: Tuples in Scala are technically limited to contain a maximum of 22 elements. Therefore, it is not possible to declare events with more than 22 properties.[1] By extension this also applies to event joining: it is impossible to join events with a combined number of properties that exceeds 22—in this case the program does not compile. A workaround is reducing the number of selected properties using the `map` operator before performing the join.

[1] This is a limitation that is also shared by other Scala libraries; e.g. at the time of this writing Slick which is part of the Typesafe Stack does not allow the querying of databases with tables that have more than 22 columns, cf. `https://issues.scala-lang.org/browse/SI-7099`.

Listing 5.1: Difference between EScala and CEScala regarding the || operator

```
var test = 0
val ev1 = new ImperativeEvent[Int]
val ev2 = ev1 && true
val ev3 = ev1 && true
val e4 = ev2 || ev3
val r1 = (e: Int) => test += e
e4 += r1
ev1(1)
ev1(2)
println(test) // equals 3 with EScala events but
              //        6 with CEScala events
```

5.2 Benchmarking CEScala

In the following sections we describe how we measure the performances of the relevant libraries regarding the test setup and present the test result and our interpretation thereof.

5.2.1 Experimental Methodology

To assess the performance of CEScala, we have conceived several tests. All tests measure the time (in milliseconds) spent for the execution of the test. The measurements were conducted on a desktop computer with an i7-4770K processor (clocked at 4.5 GHz) and 16GB of memory. Libraries included in the comparison are:

- EScala
- EsperSubscriber—Esper executed directly, with reactions attached via `subscriber` objects
- EsperListener—Esper executed directly, with reactions attached via `UpdateListeners`
- CEScalaSeparate—our integration of EScala and Esper with separate event streams
- CEScalaMerged—our integration of EScala and Esper with merged event streams

For performing the measurements we have implemented our own testing routine that takes care of executing the tests in separate VMs, warming up the VMs, running the measurements multiple times, and aggregating the results. As the key performance indicator we calculate the average of 36 measurements including confidence intervals. This procedure corresponds to the recommended way of testing the performance of Java applications [21].

The first test creates n different events, attaches to each of them a reaction, and triggers each event 100 times. The second test creates 100 different events, attaches to each of them a reaction, and triggers each event n times. Both tests are run in two variants, using only one event property and using two event properties, and in four different scenarios: $n = 100$, $n = 200$, $n = 400$, and $n = 800$.

5.2.2 Results

We show the outcomes of the tests in Tables 5.1, 5.2, 5.3, and 5.4. We plot them in Figures 5.1, 5.2, 5.3, and 5.4.

The differences between the performances of the tests with one event property and two event properties are negligible for EScala and both Esper implementations. However, CEScalaSeparate and CEScalaMerged perform slightly slower in the tests with two event properties. The reason for these results is the overhead of converting event properties to tuples in CEScala which is not needed with only one event property.

The performances of all tested libraries scale proportionally to the number of total event occurrences (the number of events multiplied with the number of times each event is triggered), with minor deviations.

EScala clearly outperforms all other implementations in all tests and all test scenarios by a great margin. It is about 50–100 times faster than CEScala or EsperListener, and the performance gap is widening with a growing number of events.

EsperSubscriber and EsperListener are performance-wise very close contenders. Usually EsperListener outperforms EsperSubscriber marginally.[2] This has encouraged us to use Esper's `UpdateListener`s instead of its `subscriber` objects in the implementation of CEScala.

CEScalaSeparate is very close to EsperListener. In some cases CEScalaSeparate even performs slightly faster than EsperListener, this requires further investigation.

Comparing CEScalaSeparate with CEScalaMerged shows that, contrary to our speculation in section 4.1, merging CEScala events to fewer Esper event streams does not improve the performance but instead costs a penalty which increases with a growing number of events. Therefore, we have chosen CEScalaSeparate as our primary implementation.

The most important benchmark for CEScala is the comparison to EsperListener, because this is the comparison that a software developer who needs CEP in his application will make. Here we expected a worse performance because CEScala adds another layer of abstraction to Esper but instead we have observed only negligible differences. Due to the benefits of CEScala over Esper regarding language-level integration we consider this a justifiable trade-off.

[2] This is only true in a Scala environment. Results can differ in a Java environment.

	100 events	200 events	400 events	800 events
EScala	0.445 ± 0.012	0.726 ± 0.009	1.352 ± 0.024	2.692 ± 0.034
EsperSubscriber	25.889 ± 0.176	47.394 ± 0.262	93.228 ± 0.436	239.468 ± 3.814
EsperListener	24.705 ± 0.201	45.011 ± 0.343	87.946 ± 0.417	228.604 ± 3.231
CEScalaSeparate	24.97 ± 0.165	49.216 ± 0.173	97.042 ± 0.349	214.793 ± 2.803
CEScalaMerged	31.827 ± 0.176	62.266 ± 0.303	124.7 ± 0.696	308.173 ± 11.44

Table 5.1: Performances of n events having one property triggered 100 times (with 95% confidence intervals)

	100 events	200 events	400 events	800 events
EScala	0.452 ± 0.003	0.744 ± 0.008	1.41 ± 0.027	2.842 ± 0.037
EsperSubscriber	24.796 ± 0.224	45.203 ± 0.201	89.325 ± 0.371	221.807 ± 2.888
EsperListener	23.488 ± 0.15	43.09 ± 0.259	84.403 ± 0.396	206.914 ± 4.557
CEScalaSeparate	31.696 ± 0.282	58.789 ± 0.392	116.497 ± 1.856	229.514 ± 0.876
CEScalaMerged	38.531 ± 0.238	72.445 ± 0.305	144.386 ± 0.627	468.173 ± 3.017

Table 5.2: Performances of n events having two properties triggered 100 times (with 95% confidence intervals)

	100 times	200 times	400 times	800 times
EScala	0.457 ± 0.025	0.775 ± 0.01	1.52 ± 0.025	2.898 ± 0.036
EsperSubscriber	25.894 ± 0.212	33.862 ± 0.243	57.054 ± 0.264	112.191 ± 0.533
EsperListener	24.447 ± 0.187	33.074 ± 0.161	56.92 ± 0.315	113.34 ± 1.659
CEScalaSeparate	24.93 ± 0.137	33.781 ± 0.114	51.46 ± 0.301	87.192 ± 0.496
CEScalaMerged	31.897 ± 0.11	42.6 ± 0.246	64.121 ± 0.293	108.643 ± 0.624

Table 5.3: Performances of 100 events having one property triggered n times (with 95% confidence intervals)

	100 times	200 times	400 times	800 times
EScala	0.457 ± 0.006	0.819 ± 0.017	1.56 ± 0.024	2.972 ± 0.033
EsperSubscriber	24.962 ± 0.227	32.696 ± 0.174	56.185 ± 0.213	112.581 ± 0.953
EsperListener	23.496 ± 0.178	32.319 ± 0.13	57.272 ± 0.746	114.536 ± 0.603
CEScalaSeparate	31.719 ± 0.153	41.376 ± 0.21	64.851 ± 0.371	111.282 ± 0.642
CEScalaMerged	38.638 ± 0.312	51.026 ± 0.256	80.279 ± 0.429	139.71 ± 0.537

Table 5.4: Performances of 100 events having two properties triggered n times (with 95% confidence intervals)

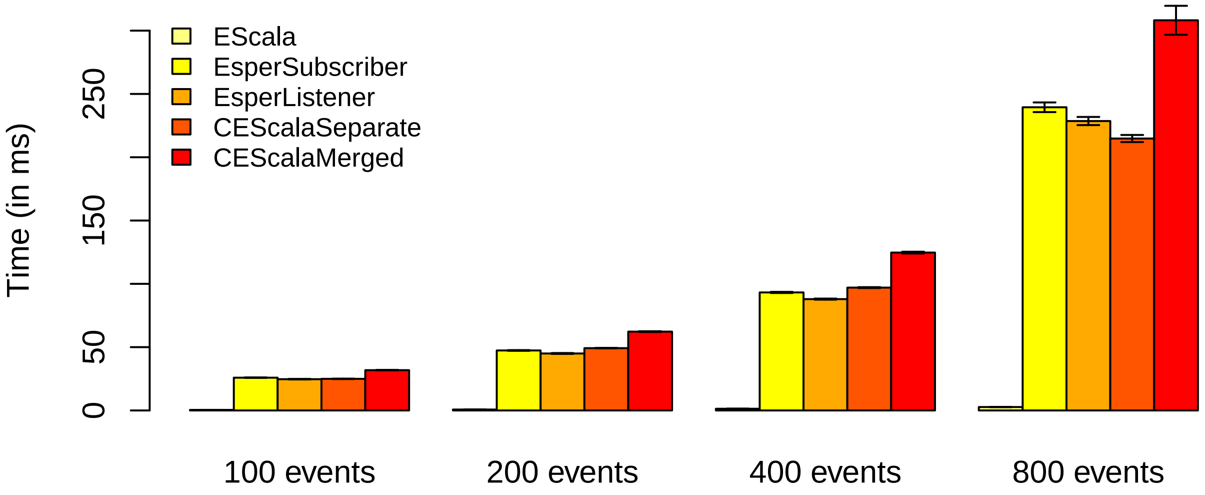


Figure 5.1: Performances of n events having one property triggered 100 times (with 95% confidence intervals)

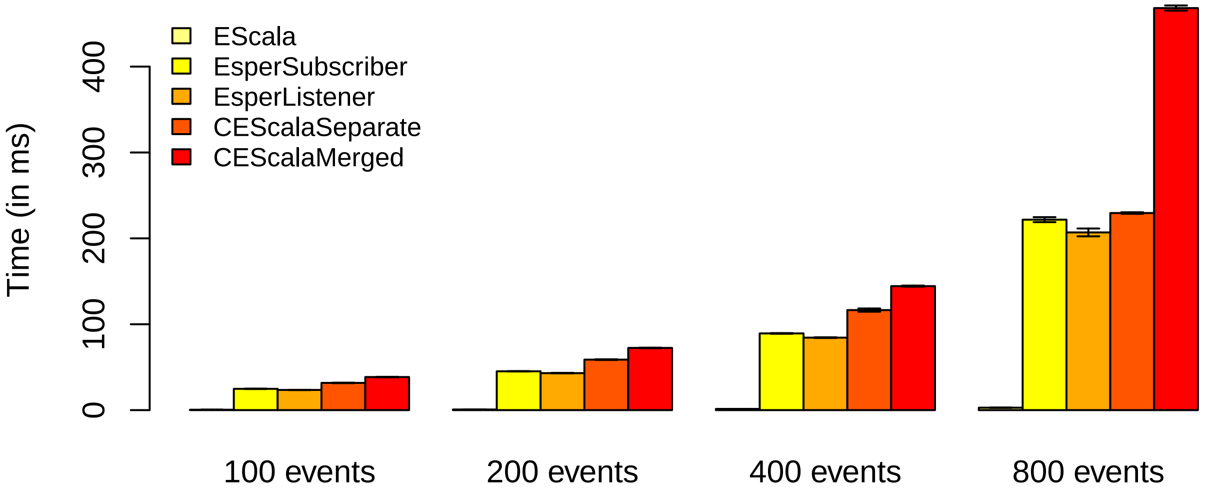


Figure 5.2: Performances of n events having two properties triggered 100 times (with 95% confidence intervals)

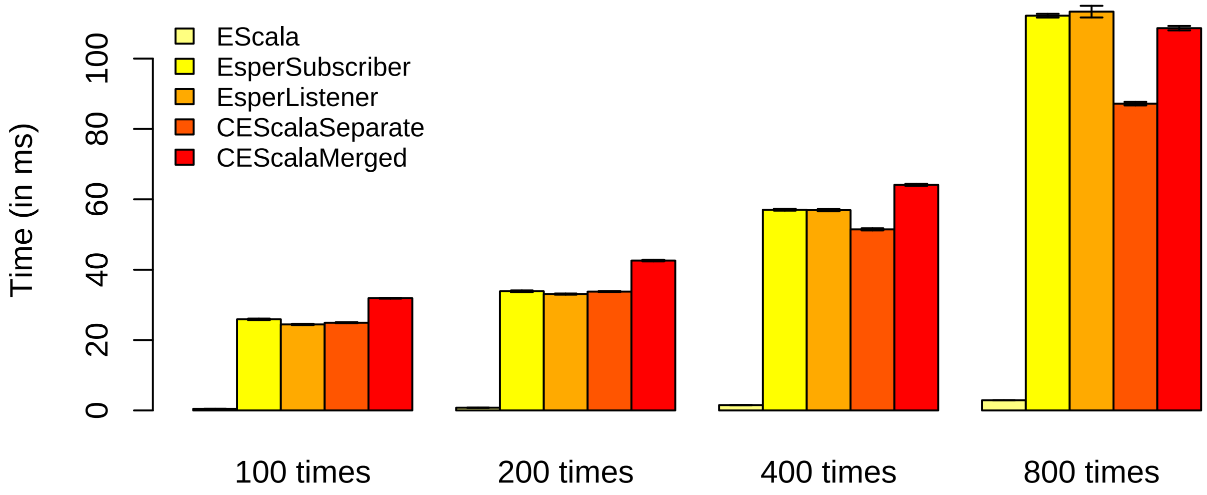


Figure 5.3: Performances of 100 events having one property triggered n times (with 95% confidence intervals)

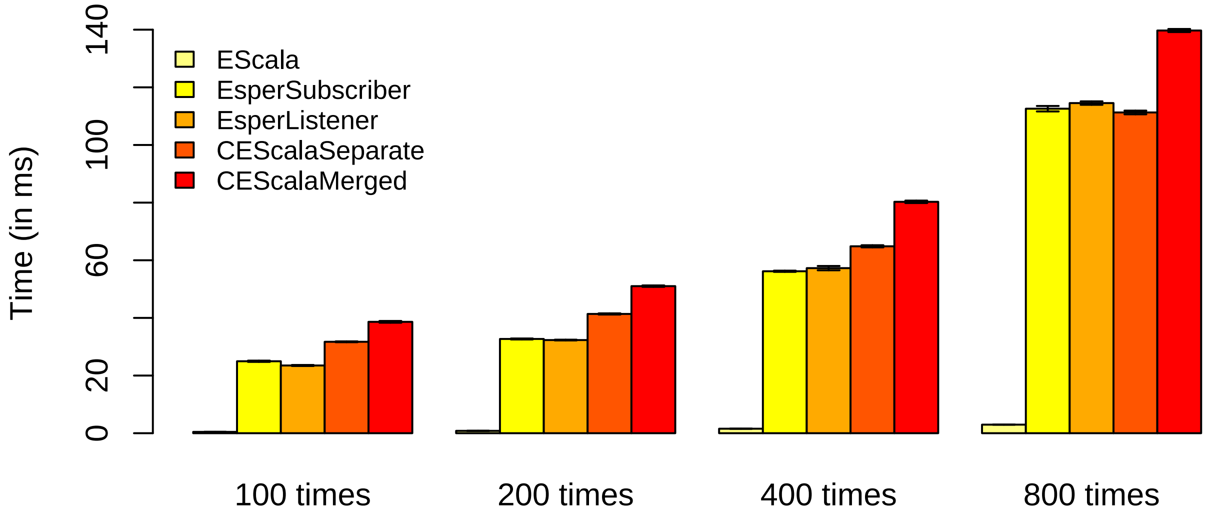


Figure 5.4: Performances of 100 events having two properties triggered n times (with 95% confidence intervals)

6 Conclusion

We have designed and provided a type-safe implementation of CEScala, a new DSL that is based on EScala and Esper. It combines the concept of imperative events found in OO designs with the expressivity that is characteristic of complex event processing libraries. In our implementation we provide a representative subset of Esper's functionality.

The performance of CEScala is almost equal to that of Esper, this means that our DSL adds only a negligible performance overhead.

However, CEScala is much slower than EScala and hence it is reasonable to use EScala instead of CEScala in applications that do not need the enhanced expressivity of CEP and in which performance plays a major role.

6.1 Future Work

Currently, CEScala only supports a subset of the features provided by Esper. Future work includes adding more operators to CEScala and thus completing the current design. Moreover, CEScala could easily be extended to support the AOP features introduced by EScala. Combining implicit events aka join points with CEP could lead to the emergence of interesting new design paradigms.

Another area of work is the support of named event properties. At present, event properties are declared using tuple types and are therefore accessed using the order of their declaration. For events with many properties it is favorable to use property names instead. The implementation could make use of a data structure from the Scala shapeless library—a Record. This construct is similar to a Map but contains heterogeneous data types.

References

[1] P. Eugster and K. R. Jayaram, “EventJava: An Extension of Java for Event Correlation”, in *ECOOP 2009 – Object-Oriented Programming: 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings*, S. Drossopoulou, Ed., ser. Lecture Notes in Computer Science, vol. 5653, Springer, 2009, pp. 570–594.

[2] V. Gasiunas, L. Satabin, M. Mezini, A. Núñez, and J. Noyé, “EScala: Modular Event-Driven Object Interactions in Scala”, in *Proceedings of the tenth international conference on Aspect-oriented software development*, ser. AOSD ’11, Porto de Galinhas, Brazil: ACM, 2011, pp. 227–240.

[3] C. Janiesch, M. Matzner, and O. Müller, “A Blueprint for Event-Driven Business Activity Management”, in *Business Process Management: 9th International Conference, BPM 2011, Clermont-Ferrand, France, August 30 - September 2, 2011. Proceedings*, S. Rinderle-Ma, F. Toumani, and K. Wolf, Eds., ser. Lecture Notes in Computer Science, vol. 6896, Springer, 2011, pp. 17–28.

[4] G. Cugola and A. Margara, “Processing Flows of Information: From Data Stream to Complex Event Processing”, *ACM Comput. Surv.*, vol. 44, no. 3, Jun. 2012.

[5] B. Peer, P. Rajbhoj, and N. Chathanur, “Complex Events Processing: Unburdening Big Data Complexities”, *Infosys Labs Briefings*, vol. 11, no. 1, pp. 53–64, 2013.

[6] T. Wormus, “Analytics and Complex Event Processing: Adding Intelligence to the Event Chain”, *Business Intelligence Journal*, vol. 13, no. 4, pp. 53–58, 2008.

[7] A. J. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. M. White, “Cayuga: A General Purpose Event Monitoring System”, in *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings*, www.cidrdb.org, 2007, pp. 412–422.

[8] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. B. Zdonik, “Aurora: a new model and architecture for data stream management”, *The VLDB Journal*, vol. 12, no. 2, pp. 120–139, Aug. 2003.

[9] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. B. Zdonik, “The Design of the Borealis Stream Processing Engine”, in *CIDR 2005, Second Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*, vol. 5, www.cidrdb.org, 2005, pp. 277–289.

[10] *Esper Reference*. [Online]. Available: `http://esper.codehaus.org/esper-4.9.0/doc/reference/en-US/pdf/esper_reference.pdf`.

[11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, “An Overview of AspectJ”, in *ECOOP 2001 – Object-Oriented Programming: 15th European Conference, Budapest, Hungary, June 18-22, 2001, Proceedings*, J. L. Knudsen, Ed., ser. Lecture Notes in Computer Science, vol. 2072, Springer, 2001, pp. 327–354.

[12] H. Rajan and G. T. Leavens, “Ptolemy: A Language with Quantified, Typed Events”, in *ECOOP 2008 – Object-Oriented Programming: 22nd European Conference Paphos, Cyprus, July 7-11, 2008 Proceedings*, J. Vitek, Ed., ser. Lecture Notes in Computer Science, vol. 5142, Springer, Jul. 2008, pp. 155–179.

[13] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, “The Many Faces of Publish/Subscribe”, *ACM Comput. Surv.*, vol. 35, no. 2, pp. 114–131, Jun. 2003.

[14] S. Suhothayan, K. Gajasinghe, I. Loku Narangoda, S. Chaturanga, S. Perera, and V. Nanayakkara, “Siddhi: A second Look at Complex Event Processing Architectures”, in *Proceedings of the 2011 ACM workshop on Gateway computing environments*, ser. GCE ’11, Seattle, Washington, USA: ACM, 2011, pp. 43–50.

[15] L. J. Fülöp, G. Tóth, R. Rácz, J. Pánczél, T. Gergely, A. Beszédes, and L. Farkas, “Survey on complex event processing and predictive analytics”, 2010.

[16] D. Spiewak and T. Zhao, “ScalaQL: Language-Integrated Database Queries for Scala”, in *Software Language Engineering: Second International Conference, SLE 2009, Denver, CO, USA, October 5-6, 2009, Revised Selected Papers*, M. G. J. van den Brand, D. Gasevic, and J. G. Gray, Eds., ser. Lecture Notes in Computer Science, vol. 5969, Springer, 2009, pp. 154–163.

[17] M. Garcia, A. Izmaylova, and S. Schupp, “Extending Scala with Database Query Capability”, *Journal of Object Technology*, vol. 9, no. 4, pp. 45–68, Jul. 2010.

[18] *An Introduction to Squeryl - A Scala ORM for SQL Databases*. [Online]. Available: `http://squeryl.org/introduction.html`.

[19] *cglib: a powerful, high performance and quality Code Generation Library*. [Online]. Available: `http://cglib.sourceforge.net`.

[20] *shapeless - An exploration of generic/polytypic programming in Scala*. [Online]. Available: `https://github.com/milessabin/shapeless`.

[21] A. Georges, D. Buytaert, and L. Eeckhout, “Statistically Rigorous Java Performance Evaluation”, in *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, ser. OOPSLA ’07, Montreal, Quebec, Canada: ACM, 2007, pp. 57–76.