Context-Oriented Programming with EventJava

K. R. Jayaram  Patrick Eugster
Department of Computer Science, Purdue University
{jayaram, peugster}@cs.purdue.edu

Abstract

Event-based computing has advanced on various fronts, especially in pervasive systems, sensor networks and large-scale asynchronous distributed systems. Recent research on Distributed Event-based Systems (DEBS) has focussed on event correlation, which is the task of processing events to identify meaningful patterns of events in the event cloud. In DEBS, software components communicate by generating, disseminating and receiving event notifications, which reify and describe the event. Several parts of an event notification are context-sensitive, depending on where the software component producing the event is deployed, the communication infrastructure available for event dissemination etc. Event Contexts may be added during the production of an event (e.g. by the runtime system executing the component) or during dissemination (by a middleware) and play an integral part in event correlation. Examples of contextual information include physical time, logical time, geographical coordinates, GPS information, information about the source of events, digital signatures, etc. EventJava [?] is an extension of Java with advanced support for event correlation. EventJava explicitly integrates the notion of event context, thereby allowing a programmer to customize the way in which events are ordered, propagated and correlated with other events. In this paper, we explain why contexts are indispensable in DEBS, present an overview of EventJava and illustrate the use of contexts through programming examples.

General Terms Languages

Keywords context-aware, events, EventJava, event correlation, streams

1. Introduction

Event-driven architecture (EDA) is a software architectural pattern promoting the design and implementation of software components that produce, detect, consume and react to (a combination of) events. An event is defined as a significant change in state in a software component. Examples of events are rainfall readings from sensors, a drop in the inventory levels of a product at a warehouse, the price of a stock, a condition detected in the network, the failure of a network link (in a network management system), a credit card transaction etc. The idiom of application event has become the cornerstone for many programming paradigms, particularly in distributed, multi-party, interaction. Publish/subscribe models and systems for instance have had a broad success in application integration, and have made their way into programming languages [?]. Several programming languages have also been proposed with event support for concurrent programming [?] or for GUI programming [?]. In pervasive computing, events are often viewed as an adequate interaction abstraction due to their asynchronous nature [?]. Other, more traditional applications of events are distributed systems monitoring (e.g., intrusion detection [?]), or active databases [?].

Event-based computing inherently decouples system components. Components communicate by generating and receiving event notifications. An event notification reifies and describes an event, and contains the data (attributes) associated with that event. Event attributes are of two types:

- Explicit attributes represent application-specific data, such as the value of a sensor, a condition detected in the network, or the value of a stock quote.
- Implicit attributes refer to contextual information. Examples are physical or logical time (e.g. simple monotonically increasing counter), geographical or logical coordinates, or event sources.

An event notification middleware decouples various components of an event-based system by delivering notifications from sources (producers, publishers) to sinks (consumers, subscribers). Sources are not a priori aware of sinks and vice versa. An event-based component is not designed to work with specific other components, thus separating communication from computation. Event Correlation (complex-event detection) is a technique for making sense of a large number of events and pinpointing the few events that are really important in that mass of information. Examples of event correlation are average of the rainfall readings from 100 sensors in a city, the occurrence of insider trading of a large volume (e.g. more than 2 million shares) of a particular stock within a 5 day period after a negative earnings report, the release of a new phone followed by 20 positive reviews in one month.

The key goal of event contexts is to make the design of software components more generic. Several parts of an event notification are dependent on the environment where a component is deployed. Certain events are timestamped w.r.t. a physical clock, e.g., stock events produced by a stock exchange, debit- and credit card transactions, flight delays etc. But, in a Distributed Event-based System (DEBS) [?], where there are multiple sources and sinks and synchronized clocks may be absent, timestamps may be based on logical clocks, e.g., Lamport clocks [?] and vector clocks [?]. The same software component may use different clocks depending on where it is deployed – physical clocks (synchronized with NTP) when deployed over a wired network and logical clocks when deployed over a wireless network. Also, events may carry security-related information, e.g., digital signatures, X.509 certificate of the producer.

[Copyright notice will appear here once ‘preprint’ option is removed.]

etc. Event contexts enable the design of sources/sinks to be separated from the software components handling authentication. Event contexts are central to the specification and detection of complex events, e.g., the choice of clocks affect the detection of patterns involving consecutive events. Hence the notion of a customizable event context is indispensable to event-based distributed programming.

EventJava promotes fully (1) asynchronous interaction seamlessly integrated with traditional synchronous method invocations, inherently unifies (2) support for uncasting as well as multicasting of events, provides increased expressiveness by supporting (3) reactions to correlation patterns and especially (4) predicates guarding those reactions. Section ?? gives an overview of EventJava. Section ?? discusses implementation issues. Section ?? describes some open avenues of research in context-oriented programming of event-based systems. Section ?? discusses closely related programming languages/models. Section ?? offers a glimpse of our ongoing research on context-aware programming with events.

2. Overview of EventJava

With EventJava, programmers do not explicitly declare events separately as data structures or “types” as for example in [?]. Instead, an event type is implicitly defined by declaring an event method, which is a specific type of asynchronous method. Complex events are defined by correlation patterns, comma-separated lists of event method headers, e.g. $e_1(), e_2(), ..., e_q()$, preceded by the keyword event, which takes the place of a return type. The formal arguments of event methods constitute the explicit attributes of the corresponding event. The implicit attributes of an event give rise to implicit arguments, which however do not have to be mentioned in the method signature and are part of the event context.

Figure ?? shows a part of an EventJava class (AccountMonitor) with two correlation patterns monitoring a bank account for suspicious overseas transactions. In the first correlation pattern, a customer is alerted if the debit card has been inactive (event method debitCardInactive) for 60 days (numdays > 60) and an overseas transaction (overseasCardTransaction) of more than $100 occurs (amount > 100). The predicate debitCardInactive < overseasCardTransaction is shorthand notation for debitCardInactive.time < overseasCardTransaction.time. The time attribute is a default implicit event attribute representing timestamps for events. Implicit event attributes are in fact fields defined globally by a context, of which an instance is passed along with every event. The code required to instantiate and pass this context is generated by our compiler. In the following simple class, an event is simply timestamped with the local physical clock. Please note that this is but a simple example, and that the notion of time is generally more complex and has to be closely aligned with the underlying communication infrastructure.

```
public class Context implements Comparable<Context>, Serializable {
  public long time;
  // more fields
  public Context() { this = System.currentTimeMillis(); }
  public Context(long time) { this.time = time; }
  public int compare(Context other) {
    if (time == other.time) return 0;
    return time > other.time ? 1 : -1;
  }
  // ...
}
```

The second correlation pattern of AccountMonitor in Figure ?? monitors the routing of money though a bank account, looking for an incoming (foreign) transfer (overseasMoneyReceipt) and an outgoing transfer (overseasMoneyReceipt) within 60 minutes of the same amount which is greater than $10,000 (conditions amount > 10000 and amount == amount). public is omitted from classes and events for brevity.

The method body (of a correlation pattern) is called a reaction and is executed asynchronously in a separate thread (typically from a thread pool) upon occurrence of an event satisfying the predicate. Arguments are considered to be values, i.e., of primitive types or conforming to Serializable, to enable event notification across address spaces. Events (event method invocations) that match the predicate are consumed by the reaction.

Consider another example – many travel agencies use farewatchers, which let customers specify destinations they are interested in, the maximum price they’re willing to pay for flight, hotel, rental car, etc. Examples are Orbitz’s “Deal Detector” and Travelocity’s “FareWatcher Plus(SM)”[5]. When customers subscribe, they decide which itineraries and destinations to track, for how long, and whether they want to be notified of price changes via email. Figure ?? illustrates a farewatcher for a customer looking for (1) an airfare less than $250 from Chicago to Miami and a hotel room for less than $100, or (2) an all inclusive weekend deal to Miami for less than $700 when the weather is expected to be warm.

An event can be notified to an instance of AccountMonitor just like a method call – a debitCardInactive(100) or a overseasMoneyReceipt(12431, 345687, 5000, "Italy"). An event, e.g., airFareDrop can also be broadcast to all instances of FareWatcher and all instances of all subclasses of FareWatcher by addressing the entire class as FareWatcher.airFareDrop("BA", "London", "Miami", 550). When an event method $e_i$ is invoked on a class $C$, it is broadcast to all live instances of $C$ and all instances of any subclass $C'$ of $C$. When the invocation happens on an interface $I$, the semantics is the same as invoking $e_i(I)$ for each class $C$ that implements $I$. By all instances of a class $C$, we mean all local instances and all remote instances of $C$ within a group. The programming model promoted by EventJava is that all event sources and sinks are within Java programs (processes). Broadcast invocation helps a source to broadcast an event to all subscribers without knowing their identities. The identity here is an object reference (possibly remote). The middleware takes care of delivering an event notification (an event method invocation) to all sinks within the group (which does limit scope). Please refer to [?] for more information related to group formation and communication.

3. Context-oriented Programming in EventJava

3.1 Geographic Coordinates

Consider a Context class using geographic coordinates in addition to timestamps. The latitude and longitude are in the decimal format, in which $0.1^\circ = 11$ km. Figure ?? shows how a correlation pattern can use this context to monitor animals in a part of a wildlife reserve. We assume that sensors are attached to animals to periodically transmit their location, vital stats etc. EventJava supports correlation over event streams through array-like indices on event methods in correlation patterns defining windows. Consider a simple pattern in AnimalMonitor1 which checks whether an animal strays out a designated safety zone. This pattern specifies the number (10) of animalLocation events being correlated, and the attributes of each of the events are accessed in the predicate and reaction body using indices. The safety zone is a circle of radius 55 km around the location of an animal.

3.2 Dynamic Contexts

Consider another pattern in AnimalMonitor2 which checks whether an animal has been at a location longer than a specific time period. This pattern requires the attribute (timestamp) to be of type Date, since the method signature requires a date object.

3.3 Distributed Contexts

The pattern in AnimalMonitor3 allows a mobile agent to store an object, which can be accessed in a correlation pattern, which is executed on a remote server.

3.4 Integration with Traditional Event Systems

EventJava can be used in conjunction with traditional event systems to enable the combined asynchronous and synchronous execution of events. This is done by using the standard CORBA event service, which provides a unified interface for both event notification and service invocation. EventJava allows the definition of complex events, which can be translated into CORBA events to enable the combined execution.

3.5 Application Scenarios

EventJava is well suited for applications that require the processing of complex events, such as financial services, telecommunication, and transportation.

3.6 Comparison with Other Event Systems

EventJava is compared with other event systems like CORBA Event Service, IBM WebSphere Event Stream, and Microsoft BizTalk. EventJava is simpler and more flexible than these systems.

3.7 Conclusion

EventJava is an alternative to traditional synchronous method invocations for event-driven systems. It provides increased expressiveness by supporting correlation patterns and predicates, and is seamlessly integrated with traditional synchronous method invocations. EventJava is well suited for applications that require the processing of complex events, and is simpler and more flexible than other event systems.

---

2. [http://www.orbitz.com/App/dealdetector/dd2_demo.jsp](http://www.orbitz.com/App/dealdetector/dd2_demo.jsp)
3. [http://www.travelocity.com](http://www.travelocity.com)
class AccountMonitor implements ...
    {
        private long debitCardNumber;
        private long account;
        private String name;
        private long SSN;
    
    event debitCardInactive(int numdays), overseasCardTransaction(float amount)
        when (debitCardInactive < overseasCardTransaction && amount > 100 && numdays > 60)
        { alertCustomer(cardnumber); }

    event overseasMoneyTransfer(long id1, long accountNum1, float amount1, String country)
        when (amount == amount1 && amount > 10000 && 60*1000 >= (overseasMoneyTransfer.time - overseasMoneyReceipt.time))
        { reportToIRS(amount, country, name, SSN, "Outgoing");
            reportMoneyRouting(account, name, SSN); }
    ...
    }

class FareWatcher implements ...
    {
        private String Address;
    
    event airFareDrop(String airline, String src, String dest, float fare),
        hotelRateDrop(String hotel, String city, String address, float rate)
        when (dest == city && dest == "Miami" && src == "Chicago" && fare <= 250 && rate <= 100)
        { Email(Address, new Deal(dest, float fare, float rate)); }

    event weekendWeatherForecast(String city, String summary, String detailed),
        lastMinuteAllInclusiveDeal(String city1, float price)
        when (city == city1 && city == "Miami" && price <= 700 && summary == "warm")
        { Email(Address, new lastMinuteDeal(city, detailed, price)); }
    ...
    }

public class Context implements Comparable<Context>, Serializable {
    public long time;
    public float latitude; // in decimal degree format
    public float longitude; // in decimal degree format
    ...
    }

class AnimalMonitor1 {
    float currLatitude;
    float currLongitude;
    2DPoint loc = new 2DPoint(currLatitude, currLongitude);
    event animalLocation[10](long animalID, String family)
        when
            for i in 0..8 animalLocation[i].animalID == animalLocation[i+1].animalID &&
                euclideanDistance(loc,new 2DPoint(animalLocation[i].latitude, animalLocation[i].longitude)) > 0.5 &&
                animalLocation[9].time - animalLocation[0].time <= 60 * 60 * 1000)
            triggerAlert("Animal " + animalID + " out of safety zone ");
        }
    ...
    }

class AnimalMonitor2 {
    float currLatitude;
    float currLongitude;
    2DPoint loc = new 2DPoint(currLatitude, currLongitude);
    event animalLocation[10](long animalID, String family)
        when
            for i,j in 0..9 animalLocation[i].animalID != animalLocation[j].animalID &&
                for i in 0..9 animalLocation[i].family.equals("Zebra") &&
                for i in 0..8 euclideanDistance(new 2DPoint(animalLocation[i].latitude, animalLocation[i].longitude),
                    new 2DPoint(animalLocation[i+1].latitude, animalLocation[i+1].longitude)) <= 0.001)
            ...
    }

Figure 1. A Bank account monitor and a farewatcher.

Figure 2. Example with geographic coordinates.
AnimalMonitor1 object. The location of an animal is treated as a (2-dimensional) point, latitude being the X-coordinate and longitude being the Y-coordinate. The distance between the animal and the AnimalMonitor1 object can thus be computed using Euclidean distance, with a distance 0.5° corresponding to 55km. An animal is considered to be outside the safety zone, and an alert is issued (triggerAlert) if there are 10 animalLocation events within a 60 minute interval where the distance between the animal and the monitor is greater than 55km.

Figure ?? also describes how groups of animals can be detected. In this example, an animal is considered to be close to another if the distance between them is less than 11m (corresponding to a euclidean distance of 0.001°). The correlation pattern in class AnimalMonitor2 looks for a group of 10 zebras.

3.2 Communication Rounds

The example on monitoring animals in Figure ?? uses timestamps based on physical clocks. In some application scenarios involving sensors, the use of physical clocks may be expensive (energy-intensive) or unnecessary. For example, if a sensor is programmed to periodically (every n units of time) broadcast values (e.g. temperature, pressure, rainfall, etc.), it may be sufficient to use communication rounds, or Lamport (logical clocks) [?]. Figure ?? sketches such a Context class, where communication rounds are obtained from a monotonically increasing counter. Communication rounds (logical clocks) induce a happened-before ordering on events from a given sensor. Assuming that 100 sensors are deployed, class SensorMonitor describes a correlation pattern which collects values from all the 100 sensors for each communication round, and computes the average value.

3.3 Vector Clocks

The Vector Clocks algorithm [?] provides partial ordering among events in a distributed system. A vector clock of a system of n processes is an array of n logical clocks, one per process, a local copy of which is kept in each process. Each process maintains a vector which is initially [0, 0, ..., 0]. Each time a process experiences an internal event, it increments its own logical clock in the vector by one. Each time a process prepares to send a message, it increments its own logical clock in the vector by one and then sends its entire vector along with the message being sent. Each time a process receives a message, it increments its own logical clock in the vector by one and updates each element in its vector by taking the maximum of the value in its own vector clock and the value in the vector in the received message (for every element) – this is abstracted by the merge method in Figure ??.

Let VC(x) and VC(y) be the vector clocks of events x and y. We say that x happens before y, or x ——> y if ∀ i: 1 ≤ i ≤ n : VC(x)[i] ≤ VC(y)[i] and ∀ j: 1 ≤ j ≤ n : VC(x)[j] < VC(y)[j]. We say that x and y are concurrent if ∃ i: 1 ≤ i ≤ n : VC(x)[i] = VC(y)[i].

Figure ?? shows a Context using vector clocks instead of physical time. The vector clock of a communicating process is maintained at the communication substrate, abstracted by the Substrate class (see Section ??). We assume that the VectorClock implements the vector clock algorithm discussed above. Project Voldemort[3] is a distributed database with an open source implementation of vector clocks, whose API is similar to VectorClock.

class Substrate {
    static VectorClock vc = new VectorClock();
    static int thisNodeID;
    void receive(Event e) {
        vc.increment(thisNodeID);
        vc.merge(e.getVectorClock());
        ...}
    }...
}

Figure 4. A Context using vector clocks.

4. Implementation

The EventJava compiler, implemented using Polyglot [?], translates EventJava programs to standard Java and generates all code required for broadcasting events, formation of groups (among sources and sinks), matching events to correlation patterns and the dispatch of reactions. Annumerable models for event-based interaction and programming have been proposed. These differ in the semantics of the very events and their handling. In particular, events can vary in the following:

S1 Representation: Events can differ in what they represent, depending on the applications in mind. Also, event models can differ in how events are reified, for example, whether events are represented as objects or method calls.

S2 Propagation: In a distributed setting, events can be propagated in various ways, with different guarantees (FIFO/total/causal order [?], reliability) and at different costs. The choice of protocol depends on the application requirements and underlying system and assumptions.

S3 Resolution: Events arriving at a given node can be assigned to handlers or reactions in various ways.

S4 Matching: In the context of correlation, event matching is another important aspect of event semantics. Sometimes a given event can match several patterns or several instances of a same pattern over time [?]. Matching can be triggered strictly upon incoming events, or also depending on local variables if permitted in patterns.

In EventJava, S?? is addressed mainly through the custom definition of explicit, but also implicit event attributes. The latter type of attributes can help convey information necessary for S?? in the form of context, which is used by the substrate responsible for propagation of events in EventJava. End-to-end guarantees on events also depend on S2 and S3. In EventJava, S2 is encapsulated by a resolver which is generated by the compiler. The matcher, also exploits the context and is tied in with the substrate. Context, substrate, and matcher are all defined as APIs which can be implemented by an advanced programmer to suit specific needs.

The Context class is verified at compilation for well-formedness. Its public fields f1, f2, ..., f9 (in the order of declaration) define

javadoc/all/voldemort/versioning/VectorClock.html

2009/4/27
public class Context implements Comparable<Context>, Serializable {
    Identifier sensorID;
    long round;
    public Context() {
        sensorID = setIdentifier();
        round = counter.incrementAndGet();
    }
    ...
    //more fields and methods
}

class SensorMonitor {
    event sensorValue[100](float value)
    when
    {
        for i in 0..98 !sensorValue[i].sensorID.equals(sensorValue[i+1].sensorID) &&
        for i in 0..98 sensorValue[i].round == sensorValue[i+1].round ) {
            float sum = 0;
            for(int j = 0 ; j < 99 ; j++) sum += sensorValue[j].value;
            float averageVal = sum/100;
            ...
            ...
    }
}

Figure 3. Sensors and rounds

the implicit event attributes. Constructors can have formal arguments corresponding to subsequences of those fields. An event method declaration can optionally list the entire context, e.g. event e() [f1, ..., fj] and an event method invocation in special cases may want to explicitly provide values for the context corresponding to a constructor, e.g. e() [f1, ..., fj] (j∈[1..q]).

5. Challenges

1. How to event contexts affect middleware that perform correlation (e.g, Hermes [?]) during the routing of events from source to sinks? Should existing routing algorithms be modified?

2. EventJava supports one context per application. Should a programming model/language support the specification of contexts unique to specific events, i.e., should the same application have many contexts, where each context corresponds to a subset of events in the application? If so, how should events with different contexts be correlated? Are there applications which require multiple contexts?

6. Related Work

To the best of our knowledge, EventJava [?] is the first language to explicitly support customizable event contexts, and integrate event contexts with event correlation. Other programming languages/models providing limited support for event correlation (but not event contexts) include (the Polyphonic C# part of) Cω [?], Join Java [?] and Scala Joins [?]. Polyphonic C# [?] supports joins (called chords) between any number of asynchronous methods (events) and atmost one synchronous method but does not support any predicates on their arguments. Consequently, matching events (method calls) to chords is straightforward. There is no support for windows of events (support for streams in Cω is not integrated with chords). Event correlation can be achieved by a staged event matching, in which a correlation pattern is matched in phases, where the occurrence of an event of a first type is a precondition for the remaining matching, which consumes that event. Staged event matching imposes an order on how events are matched to a correlation pattern. Scala Joins [?] provide Cω-like join patterns, but does not support inter-event predicates and broadcast interaction. Some event-based programming is also supported in AmbientTalk [?] through asynchronous, future-like, invocations. AmbientTalk reifies communication histories to deal with replies and exception.

Correlation is absent. A detailed comparison between EventJava and other languages is available in [?].

7. Conclusion and Future Work
