Context-Oriented Programming with EventJava*

K. R. Jayaram Patrick Eugster
Department of Computer Science, Purdue University
West Lafayette, IN 47906, USA
{jayaram,peugster}@cs.purdue.edu

ABSTRACT

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, information about the source of events, digital signatures, etc. EventJava [7], an extension of Java with advanced support for event correlation, 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 to DEBS, present an overview of EventJava and illustrate the use of contexts through programming examples.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed programming*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Classes and objects*; C.2.4 [**Computer Communication Networks**]: Distributed Systems—*Distributed applications*

General Terms

Languages

*Financially supported by National Science Foundation (NSF) through grants number 0644013 and number 0834529.

COP '09, July 7, 2009, Genova, Italy.

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 system component. Examples of events are rainfall readings from sensors, a drop in the inventory levels of a product at a warehouse, an increase in the price of a stock, the vital signs of a patient in a hospital, 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 [9]. Several programming languages have also been proposed with event support for concurrent programming [1] or for GUI programming [5]. In pervasive computing, events are often viewed as an adequate interaction abstraction due to their asynchronous nature [11]. Other, more traditional applications of events are distributed systems monitoring (e.g., intrusion detection [21]), or active databases [3].

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 counters), 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 interesting patterns of events 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.

In this paper, we introduce customizable event contexts, which are programming abstractions whose key goal is to solve the following two challenges:

- How to provide *generic* programming support for distributed event-based systems (DEBS) [17], making components deployable in different environments? This is because several parts of an event notification are dependent on the environment where a component is deployed. For example, certain events are timestamped with respect to a physical clock – stock events produced by a stock exchange, debit- and credit card transactions, flight delays etc. In 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 [15] and vector clocks [16].
- How to design a programming language that separates the design of sources/sinks from that of software components handling context-oriented information? For example, the same software component may use different notions of time depending on where it is deployed – physical clocks (synchronized with NTP[1]) when deployed over a wired network and logical clocks when deployed over a wireless network. Also, events may carry security-related information – digital signatures, X.509 certificates of the producers etc. If security information/logical time is part of the event context, it can be instantiated at runtime during the production of the event by a separate component, which is decoupled from the component producing the event.

EventJava [7] is an extension of Java that promotes fully (1) *asynchronous* interaction seamlessly integrated with traditional synchronous method invocations, inherently unifies (2) support for unicasting as well as *multicasting* of events, provides increased expressiveness by supporting (3) reactions to *correlation patterns* and especially (4) *predicates* guarding those reactions. 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.

Roadmap.

Section 2 gives an overview of EventJava. Section 3 explains how contexts can be customized in EventJava. Section 4 discusses implementation issues. Section 5 describes some open avenues of research in context-oriented programming of event-based systems. Section 6 discusses closely related programming languages/models. Section 7 concludes the paper.

2. OVERVIEW OF EventJava

With EventJava, programmers do not explicitly declare events separately as data structures or "types" as for example in [2, 6]. Instead, an event type is implicitly defined by declaring an *event method*, which is a specific type of asynchronous method. Examples of event methods are `debitCardInactive` and `overseasCardTransaction` in Figure 1. Complex events are defined by *correlation patterns*, comma-separated lists of event method headers, e.g. $e_1(),...,e_q()$, 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 1 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 suddenly an overseas transaction (`overseasCardTransaction`) of more than $100 occurs (`amount > 100`). The first predicate in the example `debitCardInactive < overseasCardTransaction` is short for `debitCardInactive.time < overseasCardTransaction.time`. The `time` attribute is an implicit event attribute representing timestamps for events. Implicit event attributes are in fact fields defined globally by a `Context` class, 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 simple `Context` class of Figure 2, 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() { time = System.currentTimeMillis(); }
  public Context(long time) { this.time = time; }
  public int compare(Context other) {
    if(timestamp == other.timestamp) return 0;
    ...
  }
  ...
}
```

Figure 2: A simple EventJava `Context`

The second correlation pattern of `AccountMonitor` in Figure 1 monitors the routing of money though a bank account, looking for an incoming (foreign) transfer (`overseasMoneyReceipt`) and an outgoing transfer (`overseasMoneyTransfer`) within 60 minutes of the same amount which is greater than $10,000 (`amount > 10000` and `amount == amount1`). **`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 events 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.

Let's 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

[1] http://en.wikipedia.org/wiki/Network_Time_Protocol

```
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 overseasMoneyReceipt(long id, long accountNum, float amount, String country),
        overseasMoneyTransfer(long id1, long accountNum1, float amount1, String country1) when
  (amount == amount1 && amount > 10000 && (overseasMoneyTransfer.time - overseasMoneyReceipt.time) <= 60*60*1000)
  {
    reportToIRS(amount, country, name, SSN, "Incoming"); //Transfers > 10,000 are reported to the IRS
    reportToIRS(amount1, country1, 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) {
           sendEmail(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") {
           sendEmail(Address, new LastMinuteDeal(city, detailed, price));
  }
  ... // Other patterns as requested by customer
}
```

Figure 1: A bank account monitor and a farewatcher.

Detector"[2] and Travelocity's "FareWatcher Plus(SM)"[3]. 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 1 illustrates a fare watcher 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 `a` 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` 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` on 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 underlying EventJava takes care of delivering an event notification (an event method invocation) to all sinks within the group(which does limit scope). Please refer to [7] for more information related to group formation and communication.

[2]http://www.orbitz.com/App/dealdetector/dd2_demo.jsp

[3]http://www.travelocity.com

3. CONTEXT-ORIENTED PROGRAMMING IN EventJava

The default context in EventJava uses timestamps based on a physical clock. In this section, we show how contexts in EventJava can be customized to handle geographic coordinates and logical clocks (Lamport clocks and vector clocks), and how attributes of the customizable `Context` can be used in correlation patterns.

3.1 Geographic Coordinates

Consider a `Context` class using geographic coordinates in addition to timestamps. The `latitude` and `longitude` are in the decimal degrees[4] format, in which $0.1^\circ \mathrel{\hat{=}} 11km$. Figure 3 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 `AnimalMonitor1` object. The location of an animal is treated as a two-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 of 0.5° corresponding to 55km. An animal is considered to be outside the safety zone, and an alert is issued (`triggerAlert`) if there

[4]http://en.wikipedia.org/wiki/Decimal_degrees

```
public class Context implements Comparable<Context> , Serializable {
  public long time;
  public float latitude; //in decimal degree format
  public float longitude; //in decimal degree format
  ... //more fields and methods
}
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 &&
      for i in 0..9 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 ");
  }
  ... //other correlation patterns and methods
}
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) { ... }
  ... //other correlation patterns and methods
}
```

Figure 3: Example with geographic coordinates

are 10 `animalLocation` events within a 60 minute interval where the distance between the animal and the monitor is greater than 55km.

The correlation pattern in `AnimalMonitor2` of Figure 3 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 3 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 (temperature, pressure, rainfall, etc.), it may be sufficient to use communication *rounds*, or Lamport clocks [15]. Figure 4 sketches such a `Context` class, where communication rounds are obtained from a monotonically increasing counter. Communication rounds (Lamport 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

Vector clocks [16] provide partial ordering among events in a distributed system. A vector clock of a system of k processes is an array of k logical clocks (l_i), one per process (p_i), a local copy ($[l_1, ..., l_k]$) of which is kept in each process. Each process maintains a vector which is initially $[0, 0..., 0]$. Each time a process (p_i) experiences an internal event, it increments its own component (l_i) in the vector by one. Each time a process (p_i) prepares to send a message, it increments its own component (l_i) 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 component 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 5.

```
class Substrate {
  static VectorClock vc = new VectorClock();
  static int thisNodeID;
  void receive(Event e) {
    vc.increment(thisNodeID);
    vc.merge(e.getVectorClock());
    ...
  }
  ...//other methods - to send messages,etc.
}
public enum Ocurred { BEFORE, AFTER, CONCURRENT }
public class Context implements ... {
  public VectorClock vc;
  public Context() {
    Substrate.vc.increment();
    vc = Substrate.vc.clone();
  }
  public Context(VectorClock vc) { this.vc = vc; }
  public Ocurred compare(VectorClock vc1) {
    return vc.compare(vc1);
  }
  ...
}
```

Figure 5: A `Context` using vector clocks.

Let $VC(x)$ and $VC(y)$ be the vector clocks of events x and y. We say that x *happens before* y, or $x \longrightarrow y$ if:

$$\forall\, i:\ 1 \leq i \leq n : VC(x)[i] \leq VC(y)[i] \quad \textit{and} \\ \exists j : 1 \leq j \leq n : VC(x)[j] < VC(y)[j]$$

We say that x and y are concurrent if:

$$\exists i : 1 \leq i \leq n : VC(y)[i] < VC(x)[i] \quad \textit{and} \\ \exists j : 1 \leq j \leq n : VC(x)[j] < VC(y)[j]$$

```
public class Context implements Comparable<Context>, Serializable {
  public Identifier sensorID;
  public long round;
  public Context() {
    sensorID = setIdentifier();
    round = counter.incrementAndGet();
  }
  ... //more fields and methods
}
class SensorMonitor {
  event sensorValue[100](float value) when
    ( for i,j in 0..99 !sensorValue[i].sensorID.equals(sensorValue[j].sensorID) &&
      for i,j in 0..99  sensorValue[i].round == sensorValue[j].round ) {
        float sum = 0;
        for(int j = 0 ; j < 99 ; j++) sum += sensorValue[j].value;
        float averageVal = sum/100;
        ...
  }
  ...
}
```

Figure 4: Sensors and rounds

Figure 5 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 4). We assume that the class `VectorClock` implements the vector clock abstraction discussed above. Project Voldemort[5] is a distributed database with an open source implementation of vector clocks, whose API is similar to `VectorClock`.

4. IMPLEMENTATION

The EventJava compiler, implemented using Polyglot [18], 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.

Innumerable 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 [15], 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 [4]. Matching can be triggered strictly upon incoming events, or also depending on local variables if permitted in patterns.

In EventJava, S1 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 S2 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 S3 and S4. In EventJava, S3 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 current implementation of our matcher (see [7, 8] for details) uses the Rete algorithm [10].

The compiler generates code to call the default constructor of `Context` when an event is produced. The `Context` class is verified at compilation for well-formedness. Its **`public`** fields $f_1, f_2, \ldots, f_q$ (in the order of declaration) define the implicit event attributes. Constructors can have formal arguments corresponding to sub*sequences* of those fields. An event method declaration can optionally list the entire context, e.g. **`event`** $e()[f_1, \ldots, f_q]$, and an event method invocation in special cases may want to explicitly provide values for the context corresponding to a constructor, e.g. $e()[f_1, \ldots, f_j]$ ($j \in [1..q]$).

5. CHALLENGES

Context-oriented programming of event-based systems is an area of active research. Two significant challenges are:

5.1 Contexts and Event Dissemination

EventJava does not require that the underlying middleware perform any event correlation. The EventJava compiler generates data structures and code required for the matching of events to correlation patterns. But some middleware like Hermes [19] perform partial event correlation (Hermes does not explicitly support streams) during the routing of events from sources to sinks. An open area of research is the interaction between event contexts and middleware that perform correlation, specifically the question of how existing routing algorithms should be modified? EventJava is a language framework, which includes a customizable communication substrate and a customizable matcher, which allows EventJava programs (with the same `Context`) to be integrated with middleware like Hermes. But, this integration is a challenge when the communicating sources and sinks use different contexts.

[5] http://project-voldemort.com

5.2 Multiple Contexts

EventJava supports one `Context` class per application. This is necessary to ensure that event correlation in EventJava preserves the ordering properties of the underlying middleware layer (for details see [7]). Let's consider an application has a set E_1 of event methods that require some notion of time (physical/logical) and another set E_2 of event methods that only require geographical coordinates. This would involve defining a single `Context` with clocks and coordinates. Should a programming model/language support the specification of contexts unique to specific events? If so, how should events with different contexts (`Context` classes) be correlated? Can aspects be used to weave contexts and event methods/correlation patterns?

6. RELATED WORK

To the best of our knowledge, EventJava [7] 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 [1], Join Java [14] and Scala Joins [12]. Polyphonic C# [1] 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, the matching of 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. Languages supporting staged event matching include Scala Actors [13] and CML [20]. Scala Joins [12] provide Cω-like join patterns, but does not support inter-event predicates and broadcast interaction. A detailed comparison between EventJava and other languages, including aspect-oriented approaches is available in [7].

7. CONCLUSION AND FUTURE WORK

In this paper, we explain why contexts are indispensable in distributed event-based systems and describe, through examples, the ways in which EventJava supports context-oriented programming. We're currently pursuing three avenues of research: (1) Extending EventJava to support multiple contexts (2) Devising annotations for flexibly configuring correlation semantics in the presence of multiple contexts, and (3) Investigating the use of domain-specific aspects to solve problems associated with multiple contexts.

8. REFERENCES

[1] N. Benton, L. Cardelli, and C. Fournet. Modern Concurrency Abstractions for C#. *TOPLAS*, 26(5):769–804, 2004.

[2] G.M. Bierman, E. Meijer, and W. Schulte. The Essence of Data Access in Cω. In *ECOOP 2005*, pages 287–311, 2005.

[3] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite Events for Active Databases: Semantics, Contexts and Detection. In *VLDB'94*, pages 606–617, 1994.

[4] S. Chakravarthy and D. Mishra. An Expressive Event Specification Language for Active Databases. *Data Knowledge Engineering*, 14(1):1–26, 1994.

[5] B. Chin and T.D. Millstein. Responders: Language Support for Interactive Applications. In *ECOOP 2006*, pages 255–278, 2006.

[6] P. Eugster. Type-based Publish/Subscribe: Concepts and Experiences. *TOPLAS*, 29(1), 2007.

[7] P. Eugster and K. R. Jayaram. Eventjava: An extension of java for event correlation. In *ECOOP 2009 (to appear)*, Available from http://krjram.googlepages.com/eventjava.pdf, 2009.

[8] P. Eugster, and K.R. Jayaram: EventJava: An Extension of Java for Event Correlation. Technical Report CSD TR #09-002, Department of Computer Science, Purdue University, http://www.cs.purdue.edu/research/technical_reports/ (2009)

[9] L. Fiege, M. Mezini, G. Mühl, and A. Buchmann. Engineering Event-Based Systems with Scopes. In *ECOOP 2002*, pages 309–333, 2002.

[10] C. Forgy. Rete: A Fast Algorithm for the Many Patterns/Many Objects Match Problem. Artificial Intelligence **19**(1) (1982) 17–37.

[11] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The *nesC* Language: A Holistic Approach to Networked Embedded Systems. In *PLDI'03*, pages 1–11, 2003.

[12] P. Haller and T. Van Cutsem. Implementing Joins using Extensible Pattern Matching. In *COORDINATION '08*, pages 135–152, 2008.

[13] Haller, P., Odersky, M.: Actors that Unify Threads and Events. In: COORDINATION 2007. 171–190.

[14] S.G. Von Itzstein and D.A. Kearney. The Expression of Common Concurrency Patterns in Join Java. In *PDPTA '04*, pages 1021–1025, 2004.

[15] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 21(7):558–565, 1978.

[16] F. Mattern. Virtual time and global states of distributed systems. In M. Cosnard, editor, *Proc. Workshop on Parallel and Distributed Algorithms*, pages 215–226, North-Holland / Elsevier, 1989.

[17] G. Mühl, L. Fiege, and P. Pietzuch. *Distributed Event-Based Systems*. Springer, 2006.

[18] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An Extensible Compiler Framework for Java. In *CC 2003*, pages 138–152, 2003.

[19] P. R. Pietzuch and J. Bacon. Hermes: A Distributed Event-Based Middleware Architecture. In *ICDCSW '02*, pages 611–618, 2002.

[20] Reppy, J.H., Xiao, Y.: Specialization of CML Message-passing Primitives. In: POPL 2007. 315–326.

[21] Trigeo. *TriGeo Security Information Manager (Trigeo SIM)*, 2007. http://www.trigeo.com/products/detailedf/.