Putting Events in Context*

Aspects for Event-based Distributed Programming

A. Holzer
Polytechnique de Montréal
Montréal, QC, Canada
adrian.holzer@polymtl.ca

L. Ziarek
Fiji Systems Inc.
Indianapolis, IN, USA
lziarek@cs.purdue.edu

K.R. Jayaram
Purdue University
West Lafayette, IN, USA
jayaram@cs.purdue.edu

P. Eugster
Purdue University
West Lafayette, IN, USA
peugster@cs.purdue.edu

ABSTRACT

Event-based programming is an appealing paradigm for developing pervasive systems since events enable the decoupling of interacting components. Unfortunately, many event-based languages and systems have hardwired notions of physical or logical time and space. This limits their adaptability and target deployment environments, as pervasive systems rely on inherent interaction and interchanging of different protocols and infrastructures.

This paper introduces *domain-specific aspects* for capturing event *context*, generalizing beyond the classic time and space dimensions associated with events. Through examples, we demonstrate that our *context aspects* — conspects for short — modularize the design and implementation of event contexts, enabling code reuse, and making programs portable across infrastructures. We illustrate the benefits of conspects by using them to transparently switch protocols in two pervasive software suites implemented in EventJava: (1) a tornado monitoring system deployed on different architectures ranging from desktop x86 to embedded LEON3, and (2) a mobile social networking suite with protocols for different application scenarios.

Categories and Subject Descriptors

D.3.3 [**Software**]: Programming Languages —*Language Constructs and Features*; D.1.5 [**Software**]: Programming Techniques —*Object-oriented Programming*

General Terms

Languages

Keywords

Aspects, Events, Context

*This research is partially funded by the Swiss National Science Foundation through project number PBLAP2-127668 and by the US National Science Foundation through grants 0644013 and 0834529.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
AOSD'11, March 21–25, 2011, Pernambuco, Brazil.
Copyright 2011 ACM 978-1-4503-0605-8/11/03 ...$10.00.

1. INTRODUCTION

Distributed applications are inherently hard to program, requiring developers to walk a thin line between specializing for the targeted distributed environment and creating generic and reusable components. These concerns are exacerbated in pervasive systems which need to deal with mobility or wireless communication.

Software specialization concerns in distributed systems, however, can be mitigated through the use of adequate abstractions. For example, architectural patterns such as *publish/subscribe* or programming languages [13, 16] centered around explicit *events* yield a higher-level model of the underlying message-passing infrastructure, abstracting data representation and communication properties. By communicating *indirectly* via production/consumption of event notifications, event-based components can remain largely decoupled, separating communication from computation. More recently, there has been increased interest in raising the level of abstraction by reasoning in terms of *complex* events composed from simpler ones. Several programming languages have been extended for event *correlation* [3, 8, 12, 17, 21, 30, 34, 38]. A closely related paradigm is *stream processing*, which allows components to operate over specific sequences of events [36, 37].

Unfortunately, current languages which support complex events do not provide adequate abstraction from the underlying distributed communication protocols, thereby being tied to specific environments or infrastructures by caveat. As an example, consider events being correlated in a pervasive system. To perform simple *time*- or *order*-based correlation à-la $e_1 < e_2$ on two events e_1 and e_2 where the former must precede the latter, clearly *timestamps* of some form are required. Even in the absence of explicit operators for order-based correlation, several distributed correlation languages have recently turned to models based on order of event occurrence or reception [30, 38], which requires timestamps. With event stream models, such order is often implicit: correlating over a sequence of m events $e[0]...e[m-1]$ implies that $e[i]$ precedes $e[i+1]$ $\forall i \in [0..m-2]$.

Yet, depending on the underlying environment, clocks may be only weakly synchronized, or not at all, making logical clocks such as Lamport clocks [24] or vector clocks [25] necessary. This in turn affects the underlying communication protocols as these rely on such information to ensure temporal constraints or order. The choice of protocol is, thus, dictated by the exact deployment environment. Similarly, many distributed applications involving mobile clients are nowadays developed with an inherent notion of *local-*

ity based on positioning mechanisms (e.g., GPS, Galileo, RFID). *Location-based* publish/subscribe [11, 15, 26] is a corresponding event-based programming model, where event publications and subscriptions can be parametrized by absolute (coordinates) and relative (range) spatial criteria. The exact representation and handling of positioning information will however strongly depend on the technology at hand.

It is thus desirable to abstract time, space, and other properties commonly associated with distributed events such as authenticity, history, or confidence levels by a notion of *event context* to further separate the design and implementation of corresponding issues from the base logic of the software components. Event contexts are instantiated upon event production and considered throughout event propagation, correlation, and handling. Any of these stages may involve the execution of context-specific code, making contexts a crosscutting concern which can be modeled as *domain-specific aspects*. Such aspects allow for adapted join points for event production or event consumption and can provide a concise mechanism for events to carry contextual information *with* them to guide the application of protocols, controlled through advice.

This paper addresses the problem of modular context representation and handling in object-oriented programming languages with support for explicit events. After illustrating that the design of event contexts is a crosscutting concern in distributed systems through real-world examples, this paper makes the following technical contributions:

1. The design of domain-specific context aspects — *conspects* for short — to modularize the design and implementation of event contexts. Our proposed design is general, allowing any language with explicit event-based constructs to be extended to support conspects.

2. An implementation of conspects in EventJava [12] on two architectures including an embedded system with a real-time Java virtual machine (Java VM).

3. An empirical evaluation of two applications: (a) an event-based tornado monitoring application and (b) a software suite for mobile social networking, to demonstrate the performance benefits of using conspects to choose different event dissemination protocols.

The remainder of this paper is structured as follows: Section 2 presents background on explicit events and event contexts. Section 3 outlines features of conspects. Section 4 presents their implementation in EventJava. Sections 5 and 6 present the tornado monitoring and mobile social networking case studies respectively. Section 7 presents related work and Section 8 concludes with final remarks.

2. BACKGROUND AND MOTIVATION

In this section we outline the core abstractions of distributed event-based programming underlying EventJava [12], and illustrate the need for specific support for event context through a *tornado monitoring* application. In tornado monitoring, several natural events are measured in order to produce a valid representation of ongoing weather conditions. *Horizontal wind velocity* and *cloud motion* are examples of such events which are used to derive different indicators, such as *Storm Relative Helicity* (SRH) [32]. The application is further analyzed and elaborated in Section 6.

2.1 Event Methods

An application event type is implicitly defined in EventJava by declaring an *event method*, a special kind of asynchronous instance-level method. This is similar to other object-oriented programming languages with support for explicit events (e.g., JoinJava [21], Cω [3], SCHOOL [8]).

2.1.1 Event definitions and production

The formal arguments of an event method correspond to the (explicit) attributes of the event type. For example, the signature hVelocity(**float** veloc, **float** alt) defines the type of horizontal wind velocity events. The types of event attributes (event method arguments) are restricted to primitive or serializable types or remote references to ensure that they can be passed over the wire. An event method declaration is preceded by the **event** keyword, which differentiates between a regular, synchronous, method with **void** return type and an asynchronous event method. For instance, an interface TornadoMonitor could declare a hVelocity event and a cMotion event as follows:

```
interface TornadoMonitor {
  event hVelocity(float velocity, float altitude);
  event cMotion(float motion, float altitude);
}
```

EventJava supports the notification of an event to an individual object (unicast, 1-to-1) simply by invocation of the corresponding event method on that object/remote object proxy. For example, a horizontal wind velocity event can be notified to an instance m of TornadoMonitor simply as m .hVelocity(...). Invokers are not blocked upon event method invocation but proceed asynchronously.

EventJava further supports implicit multicast (1-to-many) interaction by reusing notation from **static** methods. For example TornadoMonitor.hVelocity(...) dispatches the event to all objects conforming to TornadoMonitor (and subtypes) within confines specified at construction of respective objects. Destinations of a multicast event receive distinct copies of all event attributes. While EventJava supports more explicitly closed multicast groups or point-to-point (1-*of*-many) communication through specific proxies, we focus on the former multicast style for its brevity.

The same kind of multicast call can be made on any class C implementing TornadoMonitor, limiting the event to all instances of C and its sub-classes. Note that we use the term *multi*cast rather than *broad*cast, as the delivery of the event to a particular object will always be subject to individual criteria of potential destination objects as we will see shortly.

2.1.2 Event handling and composition

The easiest way to react to events is to implement respective events methods. Consider the class below:

```
class TornadoMonitorImpl implements TornadoMonitor {
  event hVelocity(float veloc, float alt, long time) {...}
  event cMotion(float motn, float alt, long time) {...}
}
```

The class allows us to receive, individually, all hVelocity and cMotion events multicast as described above. By enabling the handling of *complex* events rather than only individual events, application components can be simplified and repetitive or spurious coordination, composition, and communication can be reduced. In tornado monitoring, horizon-

tal wind velocity and cloud motion can for instance be correlated to compute SRH. SRH measures, in part, the changing of directions of winds among various altitudes within a particular area relative to a storm. Class TornadoMonitorImpl2 below *joins* events of the two types defined in interface TornadoMonitor:

```
class TornadoMonitorImpl2 implements TornadoMonitor {
  event hVelocity(float veloc, float alt, long time),
    cMotion(float motn, float alt, long time) {...}
}
```

Such joins are expressed by comma-separated lists of event method *headers*. The method body, referred to simply as *reaction*, is thus "shared" among the event method headers. In a reaction, we must prefix the event attribute names by the respective event method names, e.g., hVelocity.alt/cMotion.alt, when ambiguities can otherwise arise. There is no *implicit* matching on homonymous attributes across events.

EventJava separates the expression of *which* events are composed from *how* they are composed by the use of *guards*. These are optional and their absence is interpreted as **when true**. We can extend the example above as follows

```
class TornadoMonitorImpl3 implements TornadoMonitor {
  event hVelocity(float veloc, float alt, long time),
    cMotion(float motn, float alt, long time)
  when (hVelocity.time < cMotion.time) {
    float SRH = calculateSRH(veloc, motn);
    if(SRH >= 30 && SRH < 100) triggerAlert("Weak");
    if(SRH > =100 && SRH < 300) triggerAlert("Mild");
    if(SRH >= 300) triggerAlert("Strong");
  }
}
```

to express a strategy consisting in reacting in three ways upon hVelocity events followed by cMotion events.

A guard can use regular Java operators for boolean expressions, such as negation (!) or disjunction (||). The example is simplified for presentation. For instance, timestamps would in addition have to be within some range of each other.

Event *windows* in EventJava furthermore support the composition of several events of the same type. Such windows are syntactically unified with arrays. As an example, we can declare a class TornadoMonitorImpl4 which composes streams of events over a window size of 4 as follows:

```
class TornadoMonitorImpl4 implements TornadoMonitor {
  event hVelocity[4](float veloc, float alt, long time),
    cMotion[4](float motn, float alt, long time){...}
  ...
}
```

The attributes of an individual event can be indexed. For example, hVelocity[2].veloc or simply veloc[2] represents the veloc value of the third instance of hVelocity (indices start at 0 just as in arrays). Event streams imply ordering, i.e., for both event types above $\text{time}[i]<\text{time}[i+1]\ \forall i \in [0..2]$.

EventJava provides additional support for limited return values, or synchronous *handling* (not invocation) through an alternate **queue** keyword instead of **event**. These are not relevant for the following and are thus omitted.

2.2 Event Contexts

In our tornado monitoring example, both horizontal velocities and cloud motion events are timestamped.

2.2.1 Time

The class TornadoMonitorImpl3 above assumed synchronized physical clocks when verifying that hVelocity occurs before cMotion. The problem with this class is that it mixes application logic with information which is *specific* to that one representation of time. It cannot be deployed when sensors have clocks which are not synchronized, which depends on deployment, communication protocols, make, model, etc.

Selecting the appropriate representation of time is an important and difficult task when developing distributed systems. Relying on a too strong assumption (e.g. perfectly synchronized clocks) can lead to violating safety properties and thus to inconsistencies if the assumption does not hold in practice. For instance, processes can be suspected to have failed by some nodes but not by others. Inversely, an overly weak notion of time can lead to inefficient programs. Take the case of Lamport clocks [24], a lightweight approximation of real time by logical time. Lamport clocks capture all actual causality relations among events, yet can lead to many false positives, i.e., events which are considered to be ordered while they are unrelated. When ordering events issued by different interacting processes based on a combination of Lamport clocks with unique process identifiers to break ties, an event from a given process p with timestamp l can only be handled by another process p' once p' has received events with larger timestamps than l from all processes with smaller process identifiers than p [35]. This is clearly an inefficient solution which can be improved by Vector clocks [25], but can be much more drastically improved if synchronized clocks can be assumed (e.g., using NTP[1]) and/or the network is inherently synchronous. Rewriting an application for every possible type of logical or physical time that might be encountered throughout test and production deployments is onerous, especially as different notions of time also will imply different protocols (e.g., multicast).

2.2.2 Problem characterization

Time is an intrinsic attribute of events, present from their creation, throughout their transmission, composition, and handling. But the definition of time for events is only an instance of a general problem, namely that of defining their *contexts*. In modern mobile location-sensitive communication scenarios, the *origin* of an event may for instance be as relevant as their time of birth. As with time, such a *space* dimension can be physical (based on GPS, Galileo, RFID etc.) or logical (within a given building, room; based on identifier of node, process, object, thread, class etc.). Many more application-specific dimensions exist, such as a proof of authenticity (enforcing security policies), history (for debugging or audit), or confidence level (dealing with uncertainty).

Even if some context dimensions are specific to application families, it is desirable to separate contexts from base code as their exact implementations might vary across applications or deployments. At the same time, a library approach is unsatisfactory as it does not allow for verifying consistent uses of contexts. For instance, when correlating events of different types, these should have comparable contexts. Also, many actions for creating or handling context might be repetitive. In short, contexts are a cross-cutting concern of

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

event-based applications. We thus propose *domain-specific* aspects for implementing these in EventJava.

3. CONTEXT ASPECTS

This section outlines the design of *conspects* — context aspects for event-based distributed programming — first through the tornado monitoring example, and then by an example from mobile social networking.

3.1 Model Overview

We introduce first-class contexts, which resemble classes in that they can declare fields and methods. The former represent the implicit attributes that are associated with events to which a given context applies. The latter are predicates that are used in guards, or are used for refactoring code used in advice. Advice distinguishes contexts from classes and comes in four types with specific join points for describing their use and manipulation in the creation and handling phases of the events they apply to. Contexts are also responsible for selecting protocols for the respective phases. Below we further elaborate on these notions. We support single inheritance for contexts, but a unique context must exist for each event at every program point. When comparing two events during event correlation, we require that contexts be compatible, i.e., that the context associated with the latter event must be of a subtype of the former event's context.

3.2 Context Fields and Methods

Contexts are first-class constructs in EventJava, at the same level as classes. A first characteristic of our context aspects are context-specific fields, representing contextual attributes of events. More precisely, we separate the event attributes/event method arguments presented in the previous section into two kinds:

Explicit attributes denote application-defined event method arguments which typically vary for every event type. An example is the velocity argument of the hVelocity events in the tornado monitoring application.

Implicit attributes constitute the context of an event. We thus also refer to these simply as *context attributes*. The canonical example is time, asillustrated by the tornado monitoring example.

With conspects, implicit attributes are extracted from the application into a **context** declaration. Implicit attributes typically are the same for entire components or applications, within deployments. Consider the following class:

```
class TornadoMonitorImpl5 implements TornadoMonitor {
  event hVelocity(float veloc, float alt),
    cMotion(float mtn, float alt)
  when (hVelocity < cMotion) {...}
}
```

This class only contains the application logic to manipulate hVelocity and cMotion events. The time aspect of events is abstracted, and captured by the PhysicalClock context outlined in Figure 1. Such a declaration looks similar to a class declaration, but starts by the **context** keyword. The time attribute can be viewed as a "field" of the context. Let us focus on the context attributes and methods for now.

The comparison expressed previously on the time attribute (hVelocity.time < cMotion.time) is now expressed on the event

```
context PhysicalClock {
  long time;
  boolean "<"(PhysicalClock c)
    { return (time < c.time); }
  publish TornadoMonitor.hVelocity(..) ||
    TornadoMonitor.cMotion(..) {
    time = System.currentTimeMillis();
    with ReliableBroadcast(..);
  }
  ...
}

context LogicalClock {
  VectorClock vc;
  boolean "<"(LogicalClock vc1)
    { return (vc.compareTo(vc1) < 0); }
  publish TornadoMonitor.hVelocity(..) ||
    TornadoMonitor.cMotion(..) {
    with VCCausalBroadcast(..);
  }
  ...
}
```

Figure 1: Two different timestamp contexts: physical clocks vs. logical (vector) clocks

(names) themselves: hVelocity < cMotion. This comparison is translated to an "invocation" of "<" in the PhysicalClock conspect. The quotes are necessary because the method name is an operator in this case. While a comparison of two events on ordering is most commonly interpreted as relating to their logical or physical time of creation, any context attributes could be used for such a comparison with conspects. Consider the alternative context LogicalClock in Figure 1, where class VectorClock is a standard implementation[2] of vector clocks [25]. This context can be substituted for PhysicalClock without changes to the application.

In general, conspects may define any number of methods which can be used in guards. These methods can refer to the attributes of the context via the **thiscontext** keyword. Just like the standard **this** keyword, **thiscontext** can be omitted when disambiguation is not necessary. Context methods correspond typically to *binary* methods, i.e., they take a second context of the same context type as argument, and are used in guards in an infix notation. For example, equivalence verification of two contexts could be implemented as follows:

```
context EqualityContext {
  ...
  boolean equals(EqualityContext other) {...}
}
```

A guard could then simply compare two correlated events e_1 and e_2, making use of that context as follows:

```
event e1(...), e2(...) when (e1 equals e2) {...}
```

Context methods are not necessarily binary methods. Additional arguments can be added after the counterpart context argument; these have to be instantiated in the guard . Suppose we wanted to verify whether a given event occurred within a given range of another event. To that end, we can define the following simple conspect:

[2]E.g. http://project-voldemort.com/javadoc/all/voldemort/versioning/VectorClock.html.

```
context LocationContext {
  ...
  Location loc;
  boolean within(LocationContext other, Range rng) {...}
}
```

A guard could then simply compare two events e_1 and e_2, making use of that context as follows:

```
event e1(...), e2(...)
  when (e1 within(new Range(20)) e2) {...}
```

Context methods need not define a context argument; they can also relate to a single event. Context methods are declared in contexts and not for example in the application logic to increase cohesion and decoupling of application logic from contexts. Note that contexts can also define *local* methods which are used in context advice explained shortly.

3.3 Context Creation and Handling

As mentioned, one of the motivations for separating contexts from application code is to isolate and regroup context creation and handling operations and to avoid repetitive code. Typically, in the tornado monitoring example with hardwired physical clocks, any event creation would have to inquire the system clock through a method invocation like System.currentTimeMillis(...) and pass the obtained value for the time argument when issuing an event. This is still a relatively simple single operation, but in other cases, this instantiation can be more complex. Many operations on or with event contexts tend to be specific to the context attributes and also highly repetitive.

To isolate and regroup such code, conspects include four specific *categories* of event handling advice:

send is executed after a unicast event is produced but before it is sent.

publish is executed after a multicast event is produced but before it is multicast.

receive is executed after an event is received but before it is matched to any pattern.

consume is executed after an event is matched to a pattern but before it is consumed by a reaction.

As alluded to above, these advice are always executed at a very specific moment with respect to the event handling path. Their exact places of application are outlined by pointcuts which are rather simple in their structure. More precisely, there are only two types of elementary join points, namely such (a) denoting the event types that the advice (with the respective context) applies to and those (b) representing the *locus* of application. For example, a join point $C.e$ **in** C' in the case of a **publish** advice means that any event multicast $C.e(...)$ which occurs in class C' will be advised by the respective advice. In the case of a **receive** or **consume** advice, the same join point will imply that the advice is executed every time an instance of $C.e$ is matched for an instance of a class C', implying that C' is a subtype of C. In the same scenario, $C'.e$ **in** C' means that only multicasts of e on C' or subclasses of C' will be advised, and $C'.e$ applies to all classes.

Consider the examples of Figure 1. In both **context**s, we detail **publish** advice that are executed upon multicasting of hVelocity and cMotion events defined in the body of the TornadoMonitorImpl5 class. The ".." notation for the explicit event attributes means that the advice will not read or modify those. (Note that this concrete two dots syntax is distinct from the three dots used to indicate the omission of details for presentation simplicity.) The last line in the case of the physical clocks (**with** ReliableBroadcast(..);) passes the event to a protocol called "ReliableBroadcast" along with all context fields. The remaining arguments of the event method invocation are passed on without changes. The corresponding line in the case of the logical clocks indicates that the context attribute(s) will be created by the protocol layer – in this case "VCCausalBroadcast". These protocol invocations are transformed to calls to specific APIs, outlined shortly.

In terms of control flow, all advice types have the same semantics. Each advice is executed at a particular point of the event handling path, but it must explicitly invoke a protocol for the actual handling of that respective part in a dedicated **with** clause. That is, the advice can choose not to invoke such a protocol but that means that the event will not be implicitly passed on. Different protocols can be selected by a same advice. Consider the example of physical and vector clocks. Both **publish** advice invoke a broadcast protocol. These protocols must be bound at runtime for the advice to work properly. Furthermore, several **with** clauses can appear in the same advice, which allow protocols to be switched at runtime. Table 1 presents an overview of our four advice types and their duties/the protocols they invoke along with examples. A default protocol exists for each of these phases. It is invoked as **with** Default(..);. This default protocol is similar in nature to the proceed keyword used in around advice of AspectJ, except that, as mentioned, a protocol invocation in conspects is mandatory for the event to be handled further. Just like proceed, such an invocation can be both headed and followed by further statements. Since event handling is asynchronous, there is however never a return value for any of the advice.

Advice	Protocol type invoked	Protocol and library examples
send	Unicast	TCP/IP, UDP, Java RMI, MPI
publish	Multicast	Multisend, Reliable Broadcast, IP Multicast, JMS
receive	Matching	First received, most recent, Jess
consume	Threading	Single thread per object, thread pool (thread per pattern, thread per reaction)

Table 1: Overview of event instantiation and handling advice types and examples in conspects.

It is important to note that the **send** and **publish** advice play the role of "constructors" for contexts. They are responsible for ensuring that context attributes are initialized, though the use of initializers coupled directly with attribute declarations can simplify this task. In the physical clock example, we could have declared the timestamp as follows

```
context PhysicalClock2 {
  long time = System.currentTimeMillis();
  ...
}
```

leading to the same semantics as in Figure 1. This initialization constraint is necessary to keep in mind when using local context methods, i.e., refactoring code used in context advice into context methods. In an advice of types **receive** or **consume** one can assume that the attributes of a context have been initialized by either a preceding **send** or **publish** advice. An attribute which is always present in all advice is **this**, which refers to the actual object which is issuing the event (**send** or **publish** advice) or potentially consuming it (**receive** or **consume**). Its value is **null** in the former case if the event is created within a **static** method.

3.4 Mobile Social Networking Example

We showcase conspects through another example, this time chosen from the domain of mobile social networking to demonstrate other than time dimensions of event contexts.

3.4.1 Simple location-based publish/subscribe

The second, *Friend Finder*, application notifies friends of each other's nearby presence. Figure 2 presents an excerpt of the application code using EventJava and conspects. For simplicity, we omit the membership management, i.e., how users originally get acquainted and connected, and use a simple multicast on class SimpleFriendFinder; quite obviously a multicast will be limited to validated contacts. Users can get notified of friends located nearby through a graphical user interface (GUI) that will trigger the goOnline() method. This method periodically (every 5000 ms) multicasts beacons containing the name of the sender with a status set to *online* to other nodes located within a given event range eRange. Here, the range is defined and handled by the SpaceRestriction context. As we will see shortly in an advanced friend finder version, the range can also be defined by the user (through the GUI).

The SpaceRestriction context here uses a specific multicast protocol (LPSSHybrid), which uses the sender's location information and the restricted multicast range to physically restrict the propagation of events. We will further elaborate on this protocol in Section 6. The context also uses a specific matcher (omitted for brevity) which is rather trivial as the implemented publish/subscribe model does not usually exploit correlation of multiple events. The example alludes to a single-threaded delivery model, i.e., there will be one thread per SimpleFriendFinder object handling hello events. In the **receive** advice an event is only passed to the matcher if it originated within the given range (eRange) of the location of the receiving device at that time. This is achieved via the inRange predicate.

3.4.2 Advanced features

The outlined SimpleFriendFinder example corresponds to a location-based publish/subscribe as described by Meier and Cahill [26]. Many derived models (e.g., [11, 15]) associate a *time-to-live* (TTL) property with every event, or restrict location-based matches by complementing the event range eRange fixed by senders to restrict event receivers (delineating an *event space* around the sender) by a reception range rRange fixed by receivers to filter events based on sender location (*reception space*). With the sender device continuously moving, the sender location can be updated on events that have been multicast already.

Figure 3 outlines a solution in EventJava which incorporates all these features. In this example, we show how con-

```
class SimpleFriendFinder implements ... {
  String name;
  ...
  void goOnline(){ while(true){ Thread.sleep(5000);
      SimpleFriendFinder.hello("online", name);
    }
  }
  event hello(String status, String name)
  when (status=="online") { GUI.alert(name+" is: "+status); }
}

context SpaceRestriction {
  Location loc;
  Range eRange = ...
  ...
  boolean inRange(Range r) {
    double distance = Location.distance(Location.current(), loc);
    return distance < r;
  }
  publish SimpleFriendFinder.hello(..){
    loc = Location.current();
    with LPSSHybrid(..);
  }
  receive SimpleFriendFinder.hello(..) {
    if (inRange(eRange)) with LPSSMatcher(..);
  }
  consume SimpleFriendFinder.hello(..){with SingleThreader(..);}
}
```

Figure 2: Simple Friend Finder using EventJava

```
class FriendFinder extends SimpleFriendFinder {
  Range eRange;
  Range rRange;
  long TTL;
  void goOnline(){
    FriendFinder.hello("online", name)[eRange, TTL];
  }
  event hello(String status, String name)
    when (status=="online" && inRange(rRange))
  { GUI.alert(name + " status: " + status); }
}

context SpaceTimeRestriction extends SpaceRestriction{
  long time;
  long TTL;
  publish FriendFinder.hello(..)[eRange, TTL] {
    thiscontext.eRange = eRange;
    thiscontext.TTL=TTL;
    time = System.currentTimeMillis();
    loc = Location.current();
    with LPSSHybrid(..);
  }
  receive FriendFinder.hello(..) {
    if(System.currentTimeMillis()<time+TTL &&
      inRange(eRange)) with LPSSMatcher(..);
  }
  ...
}
```

Figure 3: Advanced Friend Finder with constrained event lifetime and spaces using EventJava

text can be set explicitly by the programmer, via a square brackets "[...]" notation. In the example, Event ranges and TTL values can be set individually for each event and reception ranges can also be changed dynamically. The square brackets thus convey required contextual arguments.

The inRange predicate verifies that the receiver is within

the event range in the **receive** advice and that the sender is in the reception space in the event guard. This **receive** advice also verifies if the received event is still valid before passing it to the matcher. Classes LPSSHybrid and LPSSMatcher implement APIs prescribed for implementations of protocols that are to be called from **publish** (as well as **send**) and from **receive** advice respectively. These APIs will be discussed in the next section. In addition to implementing a prescribed interface, the LPSSHybrid class is responsible for monitoring the current location of a device and sending position updates. Upon reception of such updates for a given object, the class is in charge of informing the LPSSMatcher which updates any events from the corresponding sender (identified by name) that it has previously received.

4. IMPLEMENTATION

This section outlines the implementation of conspects.

4.1 Weaving

We have implemented conspects as an extension to EventJava [12] which itself is implemented with the Polyglot extensible compiler framework [28]. The compiler generates a class for each **context** declaration, as a subtype of a root Context class, and generates code for the production, reification, multicast, reception, and filtering of events and dispatching of reactions. In contrast to contexts, events are transformed to a generic representation with class Event outlined in Figure 4 as they do not involve methods. The compile-time weaver inserts context arguments into event methods, and complements these by respective methods with normalized arguments (only Event) and adds appropriate where appropriate. A post-compile weaver for conspects using the ASM 2.0 bytecode manipulation framework to instrument compiled EventJava programs is underway.

4.2 Runtime Framework

To provide the proposed features, the implementation of conspects is strongly intertwined with the runtime *framework* underlying EventJava. Figures 5 and 6 present a high-level view of this framework and illustrates where/how advice are applied. Figure 5 presents source code examples prior to weaving and Figure 6 illustrates how an event is processed from invocation to reaction.

Upon event multicasting, a context object is created, then the body of the associated **publish** advice is executed during which a generic event object is created containing the context and its arguments. This object is passed to the communication *substrate* which takes care of remote communication. The substrate delivers all the serialized event method invocations to the *resolver*, which determines the classes (multicast) or objects (send) on which the methods were invoked, conveying these through *multicast objects* in the former case. When delivering the events to the event handling objects, the bodies of the respective **receive** advice are executed before the events are passed to the *matcher* (one instance per object) where they are typically but not necessarily stored in event queues. The matcher checks stored events for completed complex events and evaluates matching events based on the condition described in the event guard. An identified match is passed to the *handler* after the body of the **consume** advice is executed. The handler triggers the reaction to the event via a given threading

```
class Sender{
  // before invocation
  Receiver.event1(a1, ..., aN)[];
  // after invocation
}

class Receiver{
  event event1(a1, ..., aN)
    when(condition) {
    ... //reaction
      }
}

class Context1{
  type1 c1;
  ...
  typeN cN;
  publish Receiver.event1(..)[ ] in Sender{
    c1 = ...; // publish start
    ...
    cn = ...; // publish end
    with XSubstrate(..);
  }
  send Receiver.event1(..)[ ] in Sender{
    ... // send body
    with XSubstrate(..);
  }
  receive Receiver.event1(..){
    ... // receive body
    with YMatcher(..);
  }
  consume Receiver.event1(..){
    ... // consume body
    with ZHandler(..);
  }
}
```

Figure 4: The EventJava runtime framework – source code

model. The matcher may also include a garbage collection policy, or update and replace stored events.

Resolver and multicast objects represent application type-specific code generated at compilation to avoid costly calls through the Java reflection API. The substrate, matcher, and handler components are defined as an API, with the main types being Substrate, Matcher, and Handler respectively. Protocol invocations in advice correspond to subtypes of these respective API types. For example, the call to "VCCausalBroadcast" in the **publish** advice of Figure 1 will be transformed to a call to the multicast method of the Substrate interface outlined in Figure 5 on an instance of class VCCausalBroadcast implementing that interface.

The same approach is applied for matchers and handlers. For brevity, Figure 5 only outlines the Substrate and Matcher interfaces. Noteworthy in the former interface is the introduction of filters which are constructed at compilation from the predicates – one for each correlation pattern. Such a filter follows an SQL-like syntax similar to the *selectors* of JMS [20], and describes all conditions of a guard based solely on event attribute comparisons. These filters can be optionally used by a smart substrate to perform message filtering *throughout* the network [16]. (The performance of many multicast algorithms underlying location-based publish/subscribe as well as of algorithms underlying content-based publish/subscribe in wired networks typically hinge on such en-route filtering.) Conversely, Matcher allows the consumer object to be connected to it as a Guardian. The latter

```
public class Event implements Serializable {
  public String classN,
  public String eventN;
  public Context ctxt;
  public Object[] args;
  public Event(String typeN, Context c, Object ... args) {...}
  ...
}

public interface Substrate {
  public void unicast(Remote receiver, Event ev);
  public void multicast(Event ev);
  public void leave(String typeN, EventReceiver recv);
  public void join(String tN, EventReceiver recv, String filters);
  ...
}

public interface Matcher {
  public void add(Event ev);
  public void connect(Guardian guard);
  ...
}
```

Figure 5: Excerpt of API for conspects in EventJava

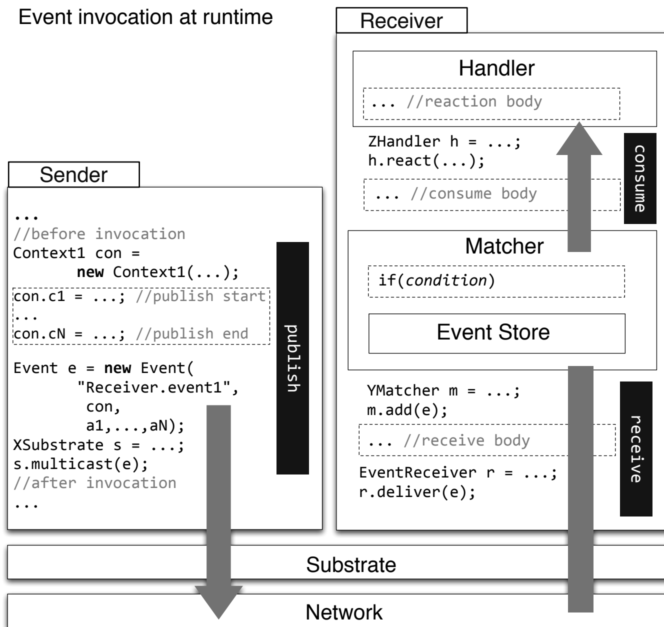


Figure 6: The EventJava runtime framework – code after weaving

interface allows application-specific guards to be invoked in a generic manner from the matcher. The code implementing this interface is generated automatically.

4.3 Fiji Virtual Machine

We ported EventJava to the Fiji VM[3] which is an ahead-of-time compiler that transforms Java bytecode into fast ANSI C code. Thus it runs on any platform that has a C compiler, threads, and locks. Supported platforms include Linux, Darwin, NetBSD, RTEMS, x86, PowerPC, and LEON. A noteworthy feature of Fiji VM is its ability to run in very restricted real-time (RT) embedded microkernels such as the RT executive for multiprocessor systems (RTEMS) [5]. RTEMS is used for hard safety-critical tasks, including ESA and NASA missions.

Fiji VM supports priority-aware locking and RT priorities for threads, and takes care not to use any unpredictable operating system (OS) facilities, like OS-provided dynamic memory allocation. Additionally, Fiji VM employs a variety of techniques, detailed in [29], for ensuring that the generated C code obeys Java semantics, and permits accurate stack maps for scanning by the garbage collector (GC). Fiji VM currently utilizes an Immix-style [4] on-the-fly concurrent RT GC.

Although the Fiji VM supports GNU classpath, the standard Java libraries are typically too big for resource constrained embedded systems. The Fiji VM thus provides Fiji Core, a smaller library, similar in scope and size to Java ME. We ported the EventJava compiler to work with Fiji Core. Additionally, since the Fiji VM is targeted at hard RT systems it does not support dynamic class loading or introspection that cannot be statically resolved using class hierarchy analysis. Since EventJava and our conspects were carefully designed *not* to necessitate such features, we were able to easily deploy in such a resource-constrained environment.

5. CASE STUDY – TORNADO MONITOR

In this section, through a tornado monitoring application, we illustrate the benefits of conspects by using them to switch between protocols without affecting the base program.

5.1 Tornado Monitoring Application

We use real-world complex events derived from NOAA's WDSS-II tornado monitoring system.[4] The system consists of a distributed tornado detection algorithm (WSR-88F TVS[5]) with several components deployed over different execution environments. Each component is tasked with processing a different set of events. The deployment architecture contains wireless field sensors over a large geographical area, each measuring atmospheric parameters like wind velocity, direction, air temperature and pressure. Wired base stations communicate with the field sensors as well as other neighboring base stations to consume and process weather-related events, thereby providing real-time tornado information. Each base station calculates several parameters like SRH, Convective Available Potential Energy (CAPE) and Energy Helicity Index (EHI). SRH is calculated by correlating streams of horizontal wind velocity and cloud motion events, and CAPE by correlating streams of air parcel temperature and ambient temperature. EHI is calculated first by finding the product of the SRH and the CAPE for a region, then the product is divided by a threshold CAPE. For evaluating the event processing throughput of our implementation, we consider 4 producers (producing the various wind speed, temperature and cloud motion events) and 6-16 consumers as described below in Section 5.3.

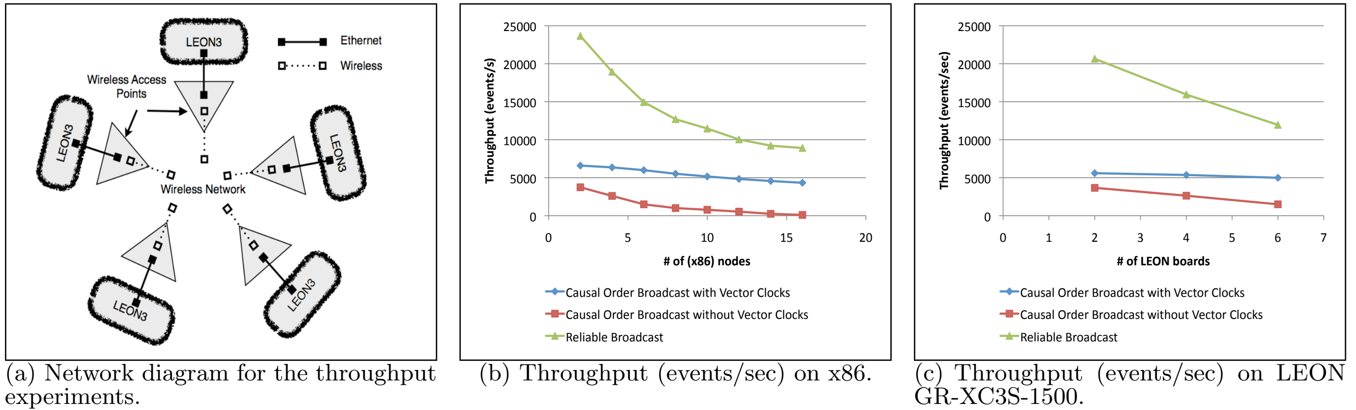
5.2 Communication Protocols

In our experiments, events were disseminated using three reliable communication protocols implemented on top of UDP:

[3] http://www.fiji-systems.com.

[4] http://www.nssl.noaa.gov/divisions/warning/swat/wdssii.php

[5] http://www.nssl.noaa.gov/divisions/warning/swat/tda.php



(a) Network diagram for the throughput experiments.

(b) Throughput (events/sec) on x86.

(c) Throughput (events/sec) on LEON GR-XC3S-1500.

Figure 7: Event processing throughput (events/sec) of the tornado monitor benchmark using three different event dissemination protocols

Reliable Broadcast: Clocks are synchronized and thus the broadcast protocol does not need to deal with ordering. Events are delivered based on their timestamps.

Vector Clock-based Causal Order Broadcast: A Causal Order Broadcast primitive ensures that messages (events) are delivered respecting cause-effect relations, thus preserving real-time relations essential for consistency. Causal Order Broadcast can be implemented with vector clocks (cf. Section 3). The use of vector clocks is enabled by our event contexts in this benchmark.

History-based Causal Order Broadcast: In this protocol, every multicast event carries a history of the events previously delivered by its source, and no process delivers an event before it delivered all antecedents. By using *Uniform* Reliable Broadcast, histories can be reduced to identifiers which in practice can be pruned regularly.

Vector Clock-based Causal Order Broadcast is preferable over History-based Causal Order Broadcast for small systems with high production rates. With low production rates and scarce network resources the latter might be favorable.

5.3 Experimental Setup

The event processing throughput of our implementation was evaluated using two different execution environments. The first was a 16 node cluster of Dell workstations, each with a Xeon 3.2GHz dual-core processor and 2 GB of RAM running Linux. The cluster was divided into one producer and 17 consumers. To adequately test our conspects in resource constrained environments, such as sensor networks, we utilize the combination of the Fiji VM + RTEMS to create realistic embedded scenarios in a 6-board cluster of GR-XC3S-1500 LEON development boards. Each board's Xilinx Spartan3-1500 field programmable gate array was flashed with a LEON3 configuration running at 40Mhz with 8MB Flash PROM and 64MB of PC133 SDRAM. This environment is outlined in Figure 7(a).

5.4 Results

Figures 7(b) and 7(c) show that the throughput of the tornado monitor application varies significantly with the choice of dissemination protocol. The presence of synchronized clocks yields the execution environment with strongest guarantees, throughput is consequently the highest with Reliable Broadcast. In the absence of synchronized clocks, typical in a distributed system, Causal Order Broadcast has to be used to compare two events. Figure 7(b) shows that the throughput of History-based Causal Order Broadcast (without vector clocks) is extremely low and degrades quickly. Using Causal Order Broadcast with vector clocks in event contexts increases the throughput on our workstation configuration by a factor of 5x in the presence of 8 consumers and by a factor of 40x in the presence of 16 consumers. Figure 7(c) shows that the use of vector clocks increases the throughput on the LEON GR-XC3S-1500 board by a factor of 3.3x in the presence of 6 consumers.

6. CASE STUDY – SOCIAL NETWORKING

In this case study, we investigate three communication (event dissemination) strategies in four different social applications running on a mobile ad hoc network. We show that choosing the optimal communication strategy can significantly improve performance in terms of message load.

6.1 Social Networking Applications

We evaluate the following application scenarios[6]:

- The *Polling* application can be used at conferences by presenters to gather votes from all participants located in a room during their talk. Participants typically receive an invitation to reply to certain questions.

- The *Friend Finder* application, which we used as example in Section 3, allows people to be notified of the presence of nearby friends. We evaluate two settings of the application, the first one with 30 users (FF1) and the second one with 300 users (FF2).

- The *Search application* is used by conference participants who want to be notified when a participant with a certain profile (e.g., name, activity, interest) comes within a certain proximity.

[6]Cf. SpotMe `http://www.spotme.com`.

- The *Advertisement* application is the last scenario we investigate. Here, shoppers are notified when nearby shoppers have offers of interest for them.

6.2 Communication Protocols

We use conspects to switch between three location-based publish/subscribe (LPSS) protocols to match events based on their content and disseminate them to interested receivers in a defined range. First, the LPSSMessageCentric protocol, which disseminates events along with contextual information in a defined range and performs matching on the receiver side. Second, the LPSSQueryCentric protocol, which relies on the propagation of queries and location criteria of receivers in a defined range to allow senders to perform the matching and initiate the routing of events. Third, the LPSSHybrid protocol, where both events and queries are broadcast up to a half the radius of the defined range and events are routed to receivers outside of this shortened radius. Each of these strategies can outperform the others depending on the communication pattern of the application scenario (e.g., number of senders and receivers, ratio between events and receivers).

6.3 Experimental Setup

We use a 500 meter-wide geographical field on which nodes evolve. The field is populated by 300 nodes which include receivers, senders and passive nodes that are running the application. Each node has WiFi capabilities with a transmission range of 50 meters. In order to simulate mobility we use the Random Waypoint (RWP) mobility model with walking speed (2-3 m/s).[7] We set a 5 second refresh rate, which indicates the time-span after which all persistent messages or all queries are re-broadcast in the message- or query-centric strategies respectively. We ran simulations with durations between 100 and 6000 seconds depending on the scenario. Parameters specific to each application are presented in Table 2: the number of *senders* and the number of *receivers*, as well as the percentage of events matching in *content* and the relevant *range* which defines the proximity filter that sender and receiver must satisfy in order for receivers to receive events. Regarding this last parameter, two different values are evaluated, a 125 meter range and a 250 meter range.

	Senders	Receivers	Cont. Match	Range
Poll	300	1	100%	125, 250m
Search	300	30	10%	125, 250m
FF1	30	30	100%	125, 250m
FF2	300	300	100%	125, 250m
Ads	30	300	100%	125, 250m

Table 2: Scenario-specific parameters

6.4 Results

The simulations presented hereafter were executed using the Sinalgo[8] network simulator, which is specifically designed for protocol simulations in wireless networks. Sinalgo replaces sockets to simulate nodes communicating wirelessly in an ad hoc network but runs on a standard Java VM. Figures 8 summarize the results of the three communication strategies for the location-based publish/subscribe service in the context of the previously described scenarios. These comparative results show the message load generated by each strategy in percentage points compared to the LPSSHybrid protocol at the end of the simulation. These ratios are expected to remain the same for longer simulations. Note that the delivery ratio of all simulations are comparable.

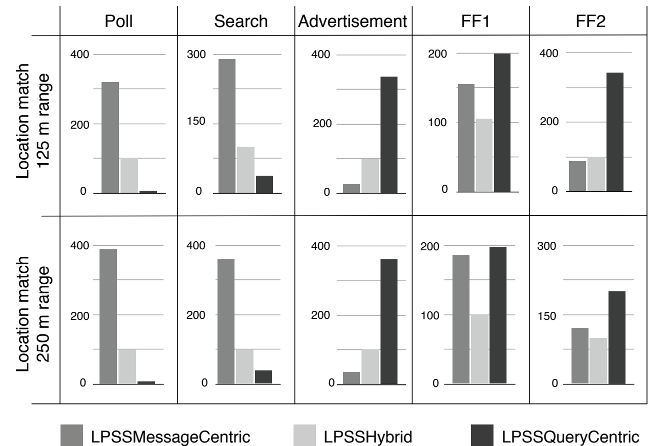


Figure 8: Message load in percent – Hybrid = 100%

Polling: Here, results show that the LPSSQueryCentric protocol outperforms the others in terms of message load. On one hand, the LPSSHybrid protocol is outperformed by a factor between 10 and 20 by the LPSSQueryCentric protocol. On the other hand, LPSSMessageCentric protocol is outperformed by a factor between 3 and 4 by the LPSSHybrid protocol. Hence the difference between the best and the worst strategy is a factor 50.

Search: Here, results show that the LPSSQueryCentric protocol outperforms the other two. The LPSSHybrid protocol is outperformed by a factor 2,5. The LPSSMessageCentric protocol is outperformed by a factor between 3 and 3,5 by the LPSSHybrid protocol. Hence the difference between the best and the worst strategy is around a factor 8.

Friend Finder I: In this application, we consider 30 participants interested in receiving notifications from each other when they are nearby. The LPSSHybrid protocol outperforms the LPSSMessageCentric protocol by a factor of 1,5 and the LPSSQueryCentric protocol by a factor of 2.

Friend Finder II: In this application, where we consider all 300 nodes as part of the friend list, the results show that the LPSSHybrid protocol and the LPSSMessageCentric protocol are close, each one beating the other by 20%. The LPSSQueryCentric protocol is outperformed by a factor between 2 and 4.

Advertisement: Here, 30 events are sent to all nodes are interested in such messages, we consider that all shopper eventually pass in front of the different shops and are

[7] RWP has been criticized [40] in the past for exhibiting a high concentration of nodes at the field center of the field and for unrealistic movement patterns. To address the first issue, we use a torus shaped map. For the second issue we consider that people moving in an open field constitutes one of the closest real-life scenarios for RWP.

[8] http://dcg.ethz.ch/projects/sinalgo/.

interested in receiving their offers. Here, the LPSSMessageCentric protocol is the most efficient in terms of message load. The LPSSHybrid protocol comes in second with between 3 and 4 times more messages, nevertheless it outperforms the query-centric by a factor 3,5. Hence, the LPSSQueryCentric protocol is outperformed by the LPSSMessageCentric protocol by a factor between 10 and 12.

As stated previously, no strategy outperforms all others in every scenario. Each strategy can demonstrate its superiority in a certain real life setting. The quantification of the differences between the message loads achieved for the different scenarios advocates for the usage of an optimal strategy if the communication pattern can be inferred in advance. If this not the case or if the communication patterns vary strongly, the LPSSHybrid protocol can be used as a safe, though not always optimal, solution, since it is never the worst of the three strategies. With conspects protocols can be switched even upon workload variations.

7. RELATED WORK

This section presents work closest related to our contexts for event-based programming, which includes efforts on models and support for context-oriented programming, aspect-oriented programming, and event-based programming.

7.1 Context-Oriented Programming

Context-oriented programming (COP) [19] is a programming paradigm which enables applications to modify their behaviors according to several dynamic program parameters. COP introduces the concept of *layers*, which modularize behavioral variations spread over several application modules. In this programming paradigm, there are two types of method definitions, namely *plain* and *layered* [1]. Layered methods consist of *base* methods containing context-independent code, and *partial* methods that contain context-dependent code. Layers can be dynamically activated by the program, potentially based on conditions in the execution environment, and can be easily composed. Base methods are executed when no layers are active, whereas partial methods are executed in the order in which layers are activated by the program. In COP side-effects generated by context-sensitive dynamically activated layers are global. Tanter [39] addresses this problem by proposing *contextual values*, which are values that depend on the context in which they are accessed and modified. An extension of Scheme with support for explicit contextual values is described in [39].

COP and the context-oriented languages mentioned above explore a completely new paradigm compared to our notion of contexts which is more specific to explicit event-based programming and pragmatic in its nature. For a more detailed description of COP, the reader is referred to [19] and [1]. Some COP extensions/incarnations are ContextS [18] for SmallTalk, or ContextJ [2] for Java.

7.2 Aspect-Oriented Programming

The popular AspectJ [23] language contains a vast collection of mechanisms which can mimic some features of conspects. For example, by introducing naming and typing conventions through compiler directives, AspectJ can support event methods. within pointcuts can determine the association between an event method call join point and the containing class, allowing for the context creation to occur similarly to **publish**/**send**. Further conventions can be introduced on the event consumer side for the expression of shared method bodies (reactions) or guards. Dynamic invocations through reflection can be used to unmarshal events received over the wire to trigger the corresponding method invocations. In general, AOP can be viewed as a form of *implicit* event-based programming as opposed to the explicit events used herein, each of which have their advantages. The inherent support for events in EventJava, especially the generation of filters which can be applied by a middleware substrate at event routing, yield important performance benefits. Without such filters, only the message-centric algorithm mentioned in Section 6.1 becomes available for location-based publish/subscribe, leading to poor performance in many scenarios as outlined.

Event-based aspect-oriented programming (EAOP) [7] provides a generic framework for the formal definition and interaction analysis of stateful aspects, with aspect composition and inter-crosscut pattern variables. Remote interaction is unfortunately not supported. Conversely, *aspects with explicit distribution* (AWED) [27] support the remote monitoring of distributed applications with remote pointcuts and distributed advice.

7.3 Event-based Programming

Many programming languages support events explicitly through some form of asynchronous methods or specific constructs. Examples include ECO [16], Ptolemy [31], AmbientTalk [6], and $Java_{PS}$ [10], or Actor-based languages and language extensions such as Erlang [9] or Scala Actors [17]. Most languages or language extensions with support for correlation, such as Polyphonic C# [3] (now integrated with $C\omega$), JoinJava [21], or SCHOOL [8], and libraries such as for Visual Basic [34], Erlang [30], or Scala [17] are based on the Join Calculus [14]. CML [33] and other languages without inherent support for correlation rely on "staged" event matching where the consumption of a first event is a precondition for subsequent matching. These languages vary in subtle but important ways in their event representation (methods, objects, functions/function calls, etc.), addressing mechanism (unicast, multicast, broadcast), or in other parts of their underlying models and implementations. Yet none of these languages support modular expression of contexts. Our own EventJava [12] was previously presented with a much more rudimentary model of contexts based on libraries and program design conventions, detailed in [22].

8. CONCLUSION

Programming pervasive systems is difficult as it requires dealing with asynchrony, distribution, and heterogeneous environments. One of the central issues in such systems is to decouple participants in time and space, as well as along other dimensions. In this paper we presented domain-specific aspects called *conspects* for capturing event context in modular way. We presented the semantics of these context aspects and illustrated their usage through two pervasive applications. We provided a detailed evaluation for the gains in performance that protocol switching based on conspects permits in two real-world case studies corresponding to those applications: a NOAA tornado monitoring application and a mobile social networking suite.

We are currently extending our work on several fronts. The major extensions consist in refining a static program

analysis for verifying that the context for a given event is unique and that contexts of different correlated events are comparable, and in post-compile weaving of conspects.

9. REFERENCES

[1] M. Appeltauer, R. Hirschfeld, M. Haupt, J. Lincke, and M. Perscheid. A Comparison of Context-oriented Programming Languages. *COP'09*.

[2] M. Appeltauer, R. Hirschfeld, and H. Masuhara. Improving the Development of Context-dependent Java Applications with ContextJ. *COP'09*, pages 1–5.

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

[4] S.M. Blackburn and K.S. McKinley. Immix: a Mark-Region Garbage Collector with Space Efficiency, Fast Collection, and Mutator Performance. *PLDI'08*, pages 22–32.

[5] OAR Corporation. *Real-Time Executive for Multiprocessor Systems (RTEMS).* http://www.rtems.org, 2009.

[6] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, and W. De Meuter. Ambient-Oriented Programming in AmbientTalk. *ECOOP'06*, pages 230–254.

[7] R. Douence, P. Fradet, and M. Südholt. Composition, Reuse and Interaction Analysis of Stateful Aspects. *AOSD'04*, pages 141–150.

[8] S. Drossopoulou, A. Petrounias, A. Buckley, and S. Eisenbach. SCHOOL: a Small Chorded Object-Oriented Language. *Electr. Notes Theor. Comput. Sci.*, 135(3):37–47, 2006.

[9] Ericsson Computer Science Laboratory. *The Erlang Pogramming Language*. www.erlang.org.

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

[11] P. Eugster, B. Garbinato, and A. Holzer. Location-based publish/subscribe. *NCA'05*.

[12] P. Eugster and K.R. Jayaram. EventJava: An Extension of Java for Event Correlation. *ECOOP'09*, pages 570–594.

[13] L. Fiege, M. Mezini, G. Mühl, and A. Buchmann. Engineering Event-Based Systems with Scopes. *ECOOP'02*, pages 309–333.

[14] C. Fournet and C. Gonthier. The Reflexive Chemical Abstract Machine and the Join Calculus. *POPL'96*, pages 372–385.

[15] D. Frey and G.-C. Roman. Context-Aware Publish Subscribe in Mobile Ad hoc Networks. *Coordination'07*, pages 37–55.

[16] M. Haahr, R. Meier, P. Nixon, V. Cahill, and E. Jul. Filtering and Scalability in the ECO Distributed Event Model. *PDSE'00*, pages 83–92.

[17] Philipp Haller and Tom Van Cutsem. Implementing Joins using Extensible Pattern Matching. *COORDINATION'08*, pages 135–152.

[18] R. Hirschfeld, P. Costanza, and M. Haupt. An Introduction to Context-Oriented Programming with ContextS. *GTTSE'07*, pages 396–407.

[19] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented Programming. *Journal of Object Technology*, 7(3):125–151, 2008.

[20] Sun Microsystems Inc. Java Message Service - Specification, version 1.1. Technical report, Sun Microsystems Inc., 2005. http://java.sun.com/products/jms/docs.html.

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

[22] K.R. Jayaram and P. Eugster. Context-Oriented Programming with EventJava. *COP'09*, pages 570–594.

[23] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W.G. Griswold. An Overview of AspectJ. *ECOOP 2001.*

[24] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, 1978.

[25] F.J. Mattern. Virtual Time and Global States of Distributed Systems. *WPDA'89*, pages 215–226.

[26] R. Meier and V. Cahill. Steam: Event-based middleware for wireless ad hoc networks. *DEBS 2002*, pages 639–644, 2002.

[27] L.D.B. Navarro, M. Südholt, W. Vanderperren, B. De Fraine, and D. Suvée. Explicitly Distributed AOP using AWED. *AOSD'06*, pages 51–62.

[28] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An Extensible Compiler Framework for Java. *CC'03*, pages 138–152.

[29] F. Pizlo, L. Ziarek, E. Blanton, P. Maj, and J. Vitek. High-level programming of embedded hard real-time devices. *EuroSys '10*, pages 69–82.

[30] H. Plociniczak and S. Eisenbach. JErlang: Erlang with Joins. *COORDINATION'10*, pages 61–75.

[31] H. Rajan and G.T. Leavens. Ptolemy: A Language with Quantified, Typed Events. *ECOOP'08*, pages 155–179.

[32] E.N. Rasmussen. Refined Supercell and Tornado Forecast Parameters. *Weather and Forecasting*, 18:530–535, 2003.

[33] J. H. Reppy and Y. Xiao. Specialization of CML Message-passing Primitives. *POPL'07*, pages 315–326.

[34] C. V. Russo. Join Patterns for Visual Basic. *OOPSLA'08*, pages 53–72.

[35] F.B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: a Tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[36] R. Soulé, M. Hirzel, R. Grimm, B. Gedik, H. Andrade, V. Kumar, and K.-L. Wu. A Universal Calculus for Stream Processing Languages. *ESOP'10*, pages 507–528.

[37] J.H. Spring, J. Privat, R. Guerraoui, and J. Vitek. StreamFlex: High-throughput Stream Programming in Java. *OOPSLA'07*, pages 211–228.

[38] M. Sulzmann, E.S.L. Lam, and P. Van Weert. Actors with Multi-headed Message Receive Patterns. *COORDINATION 2008*, pages 315–330, 2010.

[39] E. Tanter. Contextual Values. *DLS'08*, pages 1–10.

[40] J. Yoon, M. Liu, and B. Noble. Random Waypoint Considered Harmful. *INFOCOM'03.*