EventJava: An Extension of Java for Event Correlation[⋆]

Patrick Eugster and K. R. Jayaram

Department of Computer Science, Purdue University, West Lafayette, IN 47906
{peugster, jayaram}@cs.purdue.edu

Abstract. Event correlation has become the cornerstone of many reactive applications, particularly in distributed systems. However, support for programming with complex events is still rather specific and rudimentary. This paper presents EventJava, an extension of Java with generic support for event-based distributed programming. EventJava seamlessly integrates events with methods, and broadcasting with unicasting of events; it supports reactions to combinations of events, and predicates guarding those reactions. EventJava is implemented as a framework to allow for customization of event semantics, matching, and dispatching. We present its implementation, based on a compiler transforming specific primitives to Java, along with a reference implementation of the framework. We discuss ordering properties of EventJava through a formalization of its core as an extension of Featherweight Java. In a performance evaluation, we show that EventJava compares favorably to a highly tuned database-backed event correlation engine as well as to a comparably lightweight concurrency mechanism.

1 Introduction

Events demark incidents in the execution of software, a change of state in some component. In a *distributed event-based system* (DEBS), software components communicate by transmitting and receiving event notifications, which reify the events and describe the observed changes in state. Some examples of events in different domains are (*i*) the reading from a temperature sensor, (*ii*) a stock quote, (*iii*) a link failure in a network monitoring system, (*iv*) change of relationship status in a social networking tool, or (*v*) drop in inventory below a defined threshold. Interacting objects in a DEBS can act in two roles, namely as (a) *sources* (notifying events), and/or (b) *sinks* (manifesting interest in being notified of events). Event notifications describe state changes by *attributes* attached to them. *Explicit* attributes represent application-specific data; e.g, a stock quote event has the name of the organization, the price of the stock and the opening price as explicit attributes. These are sometimes paired with *implicit* attributes conveying contextual information, such as wall clock time, logical time,

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

geographical/logical coordinates, or sources. Henceforth, we will use the term *events* to refer to both the incidents underlying such events as well as to their incarnations and notifications.

Sinks are not always interested in single events. Events can be *correlated* with other events, resulting in *complex events*. Examples are:[1]

– Average of temperature readings from 10 sensors inside a boiler.
– Average of temperature readings from a sensor within a 10 minute interval.
– The price of a stock decreases for 10 successive stock quotes immediately after a negative analyst report.
– Two insider trades of a large volume ($\geq$ 10000) immediately after a stock price hits its 52 week high.
– Release of a new TV followed by 5 positive reviews in 1 month.

Event correlation is widely used in algorithmic trading and financial services, patient flow monitoring in hospitals, routing and crew scheduling in transportation, monitoring service level agreements in call centers, consumer behavior in on-line retailing and airline baggage handling, network monitoring and intrusion detection [1] just to name a few. In pervasive computing, events are often viewed as an adequate interaction abstraction due to their strongly asynchronous nature [2]. Examples of specialized event correlators in the database community are Cayuga [3], Aurora [4], and Borealis [5].

In DEBS, decoupling between the interacting objects (sources, sinks) is desired because it can lead to greater scalability. Because of this decoupling between components, interaction between them is asynchronous and often anonymous – sources and sinks do not need to know the identities of each other. Anonymous interaction is enabled by *groups* formed between sources and sinks. For example, the object that publishes stock quotes and objects which monitor stocks (at several stock brokers) are in a group – managed either using a group communication middleware (e.g. Spread [6]) or a specialized event dissemination middleware (e.g. Hermes [7], ActiveMQ [8]). The middleware is responsible for delivering events to sinks and providing fault tolerance.

In support of an increasing family of programs based on events and event-correlation in particular, we propose in this paper a novel extension of the mainstream Java programming language, called EventJava. This paper presents its design, semantics and implementation, starting by an illustration of its features through examples. The technical contributions of EventJava and this paper are:

1. An object-oriented programming model with generic support for event-based interaction. This model is implemented as an extension to Java, EventJava, incorporating features for event correlation, broadcast and unicast of events.
2. An implementation *framework* for event correlation promoting customizable propagation and matching of events. A reference implementation of this framework is presented, based on the Rete pattern matching algorithm [9]

[1] Source: www.thecepblog.com, www.complexevents.com, www.event-based.org

in the Jess expert system shell [10], and the JGroups [11] group communication system. Empirical evaluation shows that these custom off-the-shelf components can be used to achieve performance and scalability comparable to highly specialized correlation engines or lightweight concurrency mechanisms, illustrating the adequacy of the abstractions proposed in EventJava.
3. Formal semantics of event correlation in EventJava expressed as an extension to Featherweight Java (FJ) [12]. We present a *default* semantics of EventJava, where events are correlated non-deterministically, and broadcast interaction between sources and sinks does not preserve the *ordering* of events of the middleware layer. We then present the precise semantics for a more deterministic event correlation in our *reference* implementation, showing that it preserves the ordering properties of the underlying middleware layer.

The remainder of this paper is organized as follows. Section 2 presents EventJava through examples. Section 3 details its syntax and semantics. Section 4 presents an implementation of EventJava based on a framework for semantics customization. Section 5 evaluates the performance of EventJava. Section 6 explains some of our design decisions and discusses various options in EventJava. Section 7 presents related work and Section 8 draws conclusions. A companion technical report [13] provides further details such as type checking rules.

2 EventJava by Example

This section gives an overview of EventJava, introducing its features stepwise through simplified examples.

2.1 Event Methods

An application event type is implicitly defined by declaring an *event method*, a special kind of asynchronous method. The formal arguments of event methods correspond to the explicit attributes of the event type.

Handling events. Consider the example below of a travel agency which notifies its customers of severe weather in cities that are part of their flight itineraries. Instances of the `Alerts` class react to simple `severeWeather` events by retrieving the email addresses of flight passengers (in- or outbound for the city) from a database and sending emails to them. Sinks can specify additional constraints on event attributes through *predicates*, which follow the **`when`** keyword.

```
class Alerts {
  ItineraryDatabase db;
  event severeWeather(String city, String description, String source)
    when (source == "weather.com") {
      Iterator<Itinerary> it = db.getItinerariesByCity(city).iterator();
      while(it.hasNext()) { Messenger.sendEmail(it.getAssociatedEmail());}
  }
}
```

In this example, the travel agency only trusts alerts from weather.com. The method body is called a *reaction* and is executed asynchronously in a separate thread (typically from a thread pool) upon occurrence of an event satisfying the predicate. Arguments are considered to be values, i.e., of primitive types or conforming to `Serializable`, to enable event notification across address spaces. Events (event method invocations) that match the predicate are consumed by the reaction.

Unicast. Invoking an event method on an object notifies the event to that object. To that end, the source object needs a reference to the sink – a stub if the sink is remote. For example, a `severeWeather` event can be notified to an instance `a` of `Alerts` as follows:

```
a.severeWeather("Chicago", "Snow Storm 15 inches", "weather.com");
```

Broadcast. The same `severeWeather` event can be notified to all instances of `Alerts` just like a static method call:

```
Alerts.severeWeather("Chicago", "Snow Storm 15 inches", "weather.com");
```

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. By all instances of a class C, we mean all local instances and all remote instances of C within the group (see Section 4 for remote interaction and Section 6 for bootstrapping). When the invocation happens on an interface I, the event is broadcast to all instances of all classes C implementing I.

2.2 Complex Events and Correlation Patterns

Complex events are defined by *correlation patterns*, comma-separated lists of event method headers, e.g. e_1`()`,e_2`()`,...,e_q`()` , preceded by the keyword **`event`**. As we will detail later, the correlation semantics can be sensitive to order.

Consider an algorithmic trading example comparing `earningsReport` and `analystDowngrade` events. If a stock has a negative earnings report (the actual earnings per share, `epsAct`, is less than the estimate `epsEst`), followed by an analyst downgrade to "Hold", then the algorithm recommends selling the stock.

```
class StockMonitor {
  Portfolio p;
  event earningsReport(String firm, float epsEst, float epsAct, String period),
      analystDowngrade(String firm1, String analyst, String from, String to)
  when (earningsReport < analystDowngrade && firm == firm1 &&
        epsAct < epsEst && to == "Hold") {
    p.RecommendSell(firm);
  }
}
```

The first condition `earningsReport<analystDowngrade` compares an event `earningsReport` with an `analystDowngrade` event. It is a shorthand notation for `earningsReport.time < analystDowngrade.time`. The `time` attribute is a default implicit event attribute representing timestamps for events (explained shortly). `firm` can be used in lieu of `firm1` in `analystDowngrade`, but then the event name followed by the attribute must be used in the predicate and reaction for disambiguation, as in `earningsReport.firm` and `analystDowngrade.firm`. We refer to a predicate which compares attributes of two different events as a *Type-B* predicate (e.g. `firm == firm1`). Predicates which compare an attribute to a constant are referred to as *Type-A* predicates. As a comparison, our companion technical report [13] sketches a possible implementation of the same correlation pattern in standard Java.

Events that match the correlation pattern and satisfy the predicate are consumed by the reaction. Formal arguments of event methods can be similarly used in the reaction. An EventJava application developer is responsible for synchronizing accesses to shared data structures that occur *inside* the body of a reaction. One way to achieve synchronization is to add the **`synchronized`** keyword in front of the pattern; it applies to the reaction, ensuring mutual exclusion among its executions and those of other reactions and regular methods marked as synchronized.

2.3 Streams

EventJava also supports correlation over event *streams* through array-like indices on event methods in correlation patterns defining *windows*. Consider a simple pattern in fraud detection, which looks for 3 different insider trades of a stock with a combined volume $\geq$ 100000. This pattern specifies the number of `insiderTrade` events being correlated, and the attributes of each of the events are accessed in the predicate and reaction body using indices.

```
event insiderTrade[3](String firm, String name, String role, float price, long vol)
  when (insiderTrade[0].name != insiderTrade[1].name &&
    insiderTrade[1].name != insiderTrade[2].name &&
    insiderTrade[0].name != insiderTrade[2].name &&
    insiderTrade[0].firm == insiderTrade[1].firm == insiderTrade[2].firm &&
    insiderTrade[0].vol + insiderTrade[1].vol + insiderTrade[2].vol >= 100000){...}
```

A pattern of the form **`event`** e`[`n`]` **`when`** `(`p`) ...` specifies that n events e are correlated such that:

- $\forall\ i, j \in \{0, ..., n-1\}$ $i < j$ implies e`[`i`].time <` e`[`j`].time`, for example with e=`insiderTrade` above.
- Although e`[`i`].time <` e`[`i+1`].time`, the n events need not be consecutive. For example, there can be another event e'=`insiderTrade(...)` which does not satisfy predicate p such that e`[0].time <` e' `.time <` e`[1].time`. If needed, windows of consecutive events can be achieved with additional predicates e.g. based on monotonically increasing counter values assigned as attributes to events of the same type.

Aggregated events can of course be correlated with non-aggregated ones. Consider the following algorithmic trading scenario which seeks a stock decreasing monotonically in value for 10 quotes after an analyst downgrade.

```
event analystDowngrade(String firm1, String analyst, String from, String to),
    stockQuote[10](String name, float sPrice)
  when (analystDowngrade < stockPrice[0] &&
    for i in 0..8 stockQuote[i].sPrice > stockQuote[i+1].sPrice  &&
    for i in 0..9 stockQuote[i].name == analystDowngrade.name) {...}
```

A declaration e`()` without window is in fact simply treated like e`[1]()`. Consider implementing the TV example from Section 1: *Release of a new TV followed by 5 positive reviews in 1 month.* We assume that on a scale of 0 to 5, a rating above 3.5 is considered positive:

```
event tvRelease(String model, float price, String date),
  tvReview[5](String model1, File review, float rating) when
    (for i in 0..3 tvReview[i].model1 == tvReview[i+1].model &&
     for i in 0..4 tvReview[i].rating >= 3.5 &&
     tvReview[4].time - tvReview[0].time = 30*24*60*60*1000 &&
     tvReview[0].model1 == tvRelease.model) {...}
```

2.4 Matching Semantics

Event correlation semantics have different parameters [14,15]. For instance, a pattern **event** e_1`()`, e_2`()` can be matched by a sequence of events e_1^1, e_1^2, e_2^1 either as $\langle e_1^1, e_2^1\rangle$ (FIFO) or $\langle e_1^2, e_2^1\rangle$ (LIFO). In the latter case, one might even want to discard the superseded e_1^1. Different semantics can be of interest for different settings. In tightly coupled concurrency scenarios, the latter suggestion of discarding an event without consuming it seems wrong. In systems with dynamically joining and leaving participants and in the presence of predicates, it becomes infeasible in general to ensure that any event is consumed at least by one object, and obsolescence of events might be part of the application semantics.

To be able to accommodate various application types, matching in EventJava is implemented as part of a framework explained more in the following sections. Our *default* semantics presented in the next section are non-deterministic in that in the above example either outcome is possible. This reflects many concurrency settings where non-determinism is desired to achieve some form of fairness. The semantics of our *reference* implementation strike a balance between (a) static settings, i.e., where by design and deployment every event is assured to be consumed by at least one object (possibly by omitting predicates), and (b) dynamic distributed settings. They will be presented in Section 4.2.

2.5 Context

Events can have explicit and implicit attributes. In EventJava, implicit attributes form a *context*. The timestamps (`*.time`) used in Section 2.2 are but an example – though an important one. The ordering underlying our matching semantics rely on this notion.

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 following simple class, an event is simply timestamped with the local physical clock. Please note that this is but a simple example, and that the notion of time is generally more complex and has to be closely aligned with the underlying communication infrastructure and the other parts of the framework (see Section 4).

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

The `Context` class is verified at compilation for well-formedness. Its **`public`** fields f_1,f_2,...,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`, ..., `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`, ..., `f_j`]` ($j \in [1..q]$). Consider a `Context` class using geographic coordinates in addition to timestamps. The following example shows how a correlation pattern can use this context to collect rainfall readings from twenty different sensors located in a square region (see Figure 1: 55km North to 55km South and 55km East to 55km West) around the current location (which is denoted by C). The `latitude` and `longitude` are in the decimal degrees[2] format, in which $0.1° \hat{=} 11km$. Rainfall readings aggregated by the pattern should be within a 60 minute interval.

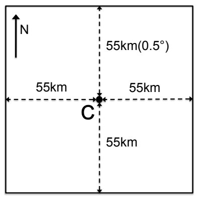

Fig. 1. Example with geographic coordinates

```
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 WeatherStats {
  float currLatitude;
  float currLongitude;
  event rainfall[20](float rainInMM, String place, int sensorID) when
    (for i in 0..19 Math.abs(rainfall[i].latitude - currLatitude) == 0.5 &&
     for i in 0..19 Math.abs(rainfall[i].longitude - currLongitude) == 0.5 &&
     for i in 0..18 rainfall[i].sensorID != rainfall[i+1].sensorID &&
     for i in 0..19 rainfall[i].time - currTime == 60 * 60 * 1000) {
      float sum = 0;
      for(int j = 0 ; j < 20 ; j++) sum += rainfall[j].rainInMM;
      float averageRainfall = sum/20;
      ...
  }
}
```

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

3 EventJava Syntax and Semantics

This section presents the syntax and semantics of EventJava in more detail, as an extension Featherweight Java (FJ) [12], dubbed Featherweight EventJava (FEJ). FEJ supports illustration, and reasoning about subtyping, inheritance, and matching semantics.

3.1 Featherweight EventJava (FEJ) Syntax

The major additions of EventJava to Java are reflected in Figure 2. As in FJ, **this** is a special variable x, and $\overline{o}$ represents a sequence $o_1 ... o_q$; separating symbols – if any – depend on the context. A given element of a sequence is referred to as o_j. Two-level nested sequences $\overline{\overline{o}}$ are also possible; in this case, an individual element can be referred to as o_j^k. The first bar above the o relates to the subscript index j, whereas the second one refers to the superscript k; $\overline{o^k}$ thus unambiguously represents $o_1^k ... o_q^k$. We use $(o)_{1..q}$ instead of $\overline{o}$ to explicitly specify the size q of $\overline{o}$.

In FEJ, a program is a parallel execution of threads, where each thread is of the form $\mathtt{T}^i(\overline{t}_;)$; the parallel composition operator $\|$ is commutative and associative. In $\mathtt{T}^i(\overline{t}_;)$, i represents a unique identifier (not necessarily continuously assigned). Threads can be created explicitly ($\mathbf{new}\ \mathtt{T}(\overline{t}_;)$) or by the system. Types (T) encompass classes (C), immutable classes (I), and value classes (D) which reflect primitive types. B, I, F, S for instance refer to booleans, integers, floats, and strings respectively. Instances of immutable and value classes are the only permissible terms for event attributes (N). Immutable classes are introduced to abstract serialization and avoid costly cloning. No assignments can occur to fields of such objects, and their fields have to be recursively immutable. (EventJava applies a simple static analysis to attempt to infer immutability and reverts to cloning if it fails.) Immutable classes cannot define patterns. FJ's call-by-value semantics are retained but as in other extensions (e.g. [16]), we introduce field assignments ($t.x := ...$) and thus $\mathbf{new}\ A(...)$ terms evaluate to locations $l(A)$ in memory. The latter terms are not used explicitly in programs; when not germane to the discussion, the type A will be omitted for brevity.

Correlation patterns include a sequence of events E, a predicate, and a reaction. Events can either declare their context ($e[n](\overline{N}\ \overline{x})[\overline{N'}\ \overline{x'}]$), or omit it ($e[n](\overline{N}\ \overline{x})$). A predicate is a conjunction or disjunction ($_{b\text{-}op}$) of simpler predicates. Among those are comparisons of value objects ($_{v\text{-}op}$), and universal quantification ($\mathbf{for}\ i\ \mathbf{in}[n..n]\ p$). An event e_j is always defined over a window of size $n_j \geq 1$, with n_j commonly 1. Predicates only allow u terms which represent a strict subgrammar of t, omitting for instance fields of **this**; even if type checking can ensure that a field f is of an immutable type, its value could otherwise change (by reassignment in another thread). In practice, the **final** modifier helps overcome this limitation.

3.2 Evaluation

Figure 3 presents auxiliary definitions for FEJ. We use contextual semantics to model dynamic semantics, introduced in Figures 4 and 5.

program	$Q ::= \emptyset \mid Q \parallel \mathtt{T}^i(\overline{t}\,\mathtt{;})$	value	$v ::= l(A) \mid \mathbf{new}\ D\,(d)$
mutable	$L ::= \mathbf{class}\ C\ \mathbf{extends}\ C\{\overline{T}\ \overline{f}\mathtt{;}\ K\ \overline{M}\ \overline{P}\}$	type	$T ::= C \mid I \mid D$
immutable	$J ::= \mathbf{class}\ I\ \mathbf{extends}\ I\{\overline{N}\ \overline{f}\mathtt{;}\ K\ \overline{M}\}$	class	$A ::= C \mid I$
constructor	$K ::= A\,(\overline{T}\ \overline{f})\,\{\mathbf{super}\,(\overline{f})\,\mathtt{;}\ \mathbf{this}.\overline{f}\mathtt{:=}\overline{f}\mathtt{;}\,\}$	attribute	$N ::= I \mid D$
method	$M ::= T\ m\,(\overline{T}\ \overline{x})\,\{\overline{t}\mathtt{;}\ \mathbf{return}\ t\mathtt{;}\,\}$	pattern	$P ::= \mathbf{event}\ \overline{E}\ \ \mathbf{when}\,p\,\{\overline{t}\mathtt{;}\,\}$
event	$E ::= e\,[n]\,(\overline{N}\ \overline{x})\,[\overline{N'}\ \overline{x'}] \mid e\,[n]\,(\overline{N}\ \overline{x})$		
filter	$u ::= u\,.\,f \mid (T)\,u \mid \mathbf{new}\ D\,(d) \mid x\,[n] \mid x\,[i]$		
predicate	$p ::= u\ \textit{v-op}\ u \mid p\ \textit{b-op}\ p \mid\ !\ p \mid \mathbf{for}\,i\,\mathbf{in}\,[n\,..\,n]\ p$		
term	$t ::= v \mid x \mid t\,.\,f \mid t\,.\,f\mathtt{:=}t \mid t\,.\,m\,(\overline{t}) \mid (T)\,t \mid t\,.\,e\,(\overline{t}) \mid t\,.\,e\,(\overline{t})\,[\overline{t}] \mid C\,.\,e\,(\overline{t})$		
	$\mid C\,.\,e\,(\overline{t})\,[\overline{t}] \mid \mathbf{new}\ A\,(\overline{t}) \mid \mathbf{new}\ \mathtt{T}\,(\overline{t}\,\mathtt{;})$		

$$\begin{array}{ll}
d \in \mathbb{D} & (D, d, \mathbb{D}) \in \{(\mathtt{B}, b, \mathbb{B}), (\mathtt{I}, z, \mathbb{Z}), (\mathtt{F}, r, \mathbb{R}), (\mathtt{S}, s, \Sigma^*), \ldots\} \\
l \in \mathrm{dom}(\mathcal{L}) & \textit{v-op} \in \{\mathtt{==}, \mathtt{<=}, \mathtt{<}\} \\
n \in \mathbb{Z}^+ & \textit{b-op} \in \{\mathtt{\&\&}, \mathtt{||}\}
\end{array}$$

Fig. 2. Featherweight EventJava (FEJ) syntax.

$$fields(\mathtt{Object})=\emptyset \qquad [\textsc{Obj-Class}]$$

$$CT(\mathtt{E})=\mathbf{class}\ \mathtt{E}\ \mathbf{extends}\ \mathtt{Object}\{\} \qquad [\textsc{Ev-Class}]$$

$$\frac{\begin{array}{c} CT(A)=\mathbf{class}\ A\ \mathbf{extends}\ A'\{\overline{T}\ \overline{f}\ \mathtt{;}\ \ldots\} \\ fields(A')=\overline{T'}\ \overline{f'} \end{array}}{fields(A)=\overline{T'}\ \overline{f'},\ \overline{T}\ \overline{f}} \qquad [\textsc{Field-Type}]$$

$$\frac{\begin{array}{c} CT(C)=\mathbf{class}\ C\ \mathbf{extends}\ \ldots\{\ldots\ \overline{P}\} \\ \mathbf{event}\ldots\ e\,[n]\,(\overline{N}\ \overline{x})\ \ldots\ \{\overline{t}\mathtt{;}\,\} \in \overline{P} \end{array}}{etype(e, C)=n\times\overline{N}\ \overline{\star N}\rightarrow\mathtt{E}} \qquad [\textsc{Ev-Type}]$$

$$\frac{CT(C)=\mathbf{class}\ C\ \mathbf{extends}\ C'\{\ldots\ \overline{P}\} \qquad \nexists\ldots e\ldots\in\overline{P}}{etype(e, C)=etype(e, C')} \qquad [\textsc{Ev-Type-Inh}]$$

$$\frac{\begin{array}{c} CT(C)=\mathbf{class}\ C\ \mathbf{extends}\ \ldots\{\ldots\ \overline{P}\} \\ \overline{\mathbf{event}\ e\,[n]\,(\overline{N}\overline{x})\,[\overline{N'x'}]\ \mathbf{when}\,p\,\{\overline{t}\mathtt{;}\,\}} \in \overline{P} \end{array}}{rbody(\overline{e}, C)=\ (\overline{\overline{x}\overline{x'}}, p, \overline{t})} \qquad [\textsc{React-Body}]$$

$$\frac{\begin{array}{c} CT(C)=\mathbf{class}\ C\ \mathbf{extends}\ \ldots\{\ldots\ \overline{P}\} \\ \forall\ j\ fresh\ \overline{x^j}=x^j_1\ldots x^j_q \qquad q=|\overline{\star N}| \\ \mathbf{event}\ \overline{e\,[n]\,(\overline{Nx'})}\ \mathbf{when}\,p\,\{\overline{t}\mathtt{;}\,\} \in \overline{P} \end{array}}{rbody(\overline{e}, C)=\ (\overline{\overline{x'}\overline{x}}, p, \overline{t})} \qquad [\textsc{React-Body-Def}]$$

$$\frac{\begin{array}{c} CT(C)=\mathbf{class}\ C\ \mathbf{extends}\ C'\{\ldots\ \overline{P}\} \\ \nexists\ldots\overline{e\ldots}\ldots\in\overline{P} \end{array}}{rbody(\overline{e}, C)=rbody(\overline{e}, C')} \qquad [\textsc{React-Body-Inh}]$$

Fig. 3. Auxiliary definitions for FEJ.

Contextual semantics. Event methods are typed (*etype*) just like ordinary methods ([Ev-Type], [Ev-Type-Inh]), since they don't have return values. E is used as a placeholder for **event** and has neither methods nor fields. Evaluation takes place on tuples; in the context of local evaluation $\longrightarrow$, such a tuple is a term together with an object store $\mathcal{L}$ and an event store $\mathcal{S}$ (see Figure 4). $\mathcal{S}(l)$ represents per-object queues of events of the form $(e, \overline{v}\overline{v'})\cdot(e, \overline{v}\overline{v'})\cdot\ldots$. Global evaluation $\Longrightarrow$ is similar to local evaluation, but relates programs instead of terms. $\longrightarrow^*$ is the transitive closure of $\longrightarrow$. $\mathcal{L}$ changes when vales are assigned to object fields ([Field-Ass-R]) and when new objects are created ([Loc-R]). In [Forall-R], quantification over integers $n..n'$ is reduced to $n' - n + 1$ predicates. We sometimes use $(o)_{1..q}$ instead of $\overline{o}$ to make the size q of $\overline{o}$ explicit.

Contexts and broadcast. The context is represented by a set of terms $\overline{\nabla t}$ of types $\overline{\star N}$. In an event $e\,[n]\,(\overline{N}\ \overline{x})\,[\overline{N'}\ \overline{x'}]$, we assume, without loss of generality, that the first term of the context x'_1 is used for ordering. In FEJ, values for the context variables are either specified explicitly ($t\,.\,e\,(\overline{t})\,[\overline{t}]$, $C\,.\,e\,(\overline{t})\,[\overline{t}]$) or instantiated during reduction ([Ev-Def-R, Ev-Bcast-Def-R]). To simplify the calculus rules, we make two assumptions: (i) Either all events in a given pattern declaration specify the context or omit it. (ii) In all invocations of an event method, the context is either explicitly specified or instantiated during reduction. Rules [Ev-Def-R] and [Ev-Bcast-Def-R] mimic actions performed by the runtime/middleware in an implementation. In [Ev-Def-R] and [Ev-Bcast-Def-R], for a given object l, the ∇t_1^l terms assigned must evaluate to values that are totally ordered, in increasing order. In [Ev-R], events are added to the queue corresponding to object l. Broadcast ([Ev-Bcast-Def-R, Ev-Bcast-R]) is *semantically* equivalent to a sequence of unicasts ([Ev-R]). Note that in both [Ev-Bcast-Def-R] and [Ev-Bcast-R], broadcast is not *atomic*, because it takes q reduction steps, which can be interleaved with reduction steps of other threads running in parallel. So there is no *global* total order, i.e., events in different queues may be ordered differently.

Correlation. The core of the semantics is the reaction rule [React-R], which relies on the $match()$ predicate, and uses a number of auxiliary definitions. $\pi_e \mathcal{S}(l)$ is a projection that simply extracts a subsequence of events of type e from an event queue $\mathcal{S}(l)$. Set complement $\mathcal{S}(l)\backslash(...)$ and inclusion $\in$ follow the usual intuition, but are specified to simplify understanding of the weaknesses of the present semantics and the refined semantics for our reference implementation presented in the next section. The $match()$ predicate ([Patt-Match]) simply takes *any* set of events matching any pattern defined for a given object ($\in$), regroups the corresponding events ($\mathcal{N}$) and creates a corresponding variable substitution (Θ). In case the predicate for the pattern evaluates to $true$, [React-R] simply removes the events from the queue and creates a new thread (thread pool in practice) to execute the reaction. Note that [React-R] does not produce any side-effects in $\mathcal{L}$, due to the constraints on predicates. Given the non-deterministic nature of the event matching, paired by the simple reduction of an event broadcast to a multi-send in [Ev-Bcast-R], two instances of a same class receiving identical sets of broadcast (only) events will not necessarily correlate the same events.

The matching semantics presented here are intentionally weak and serve mostly as illustration. The handling of broadcast for instance does not assume more than reliable point-to-point communication. Using expensive event dissemination protocols results in better ordering guarantees. The semantics of our reference implementation (Section 4.2) uses deterministic selection of matching events and total order broadcast to disseminate events, providing strong guarantees on the order of execution of reactions (Section 4.2). But, FEJ does not force the use of a specific dissemination protocol or protocol family, because the choice strongly depends on the application, and the underlying infrastucture and system model. Ordering guarantees, for instance, induce a sensible overhead most of the time. In specific cases, they may be achieved more easily or even spontaneously (e.g. if the basic communication mechanism is broadcast-based such as on a single Ethernet wire or in certain wireless settings) or simply not be needed. Ordering properties, just like correlation semantics in general, can not be automatically inferred from the application.

Evaluation contexts

$$\begin{array}{l} E ::= [\,] \mid (T)\,E \mid E.f \mid E.f\texttt{:=}t \mid v.f\texttt{:=}E \mid E.m\,(\overline{t}) \\ \quad \mid v.m\,(E) \mid E.e\,(\overline{t}) \mid E.e\,(\overline{t})\,[\overline{t}\,] \mid v.e\,(E)\,[\overline{t}\,] \\ \quad \mid v.e\,(E) \mid v.e\,(\overline{v})\,[E] \mid C.e\,(E) \mid C.e\,(E)\,[\overline{t}\,] \\ \quad \mid C.e\,(\overline{v})\,[E] \mid \textbf{new}\ A\,(E) \mid \overline{v}\texttt{;}E\texttt{;}\overline{t} \mid \overline{v}\texttt{,}E\texttt{,}\overline{t} \\ \quad \mid E\ \textit{v-op}\ t \mid v\ \textit{v-op}\ E \mid E\ \textit{b-op}\ p \mid v\ \textit{b-op}\ E \mid\, !\,E \\ \quad \mid E\texttt{;}\ \textbf{return}\ t \mid v\texttt{;}\ \textbf{return}\ E \end{array}$$

$$\boxed{\langle Q, \mathcal{L}, \mathcal{S}\rangle \Longrightarrow \langle Q', \mathcal{L}', \mathcal{S}'\rangle}$$

$$\langle Q \parallel \mathtt{T}^i(\overline{v}\texttt{;}), \mathcal{L}, \mathcal{S}\rangle \Longrightarrow \langle Q, \mathcal{L}, \mathcal{S}\rangle \quad \text{[Thread-Kill-R]}$$

$$\frac{j\ \mathit{fresh}}{\langle Q \parallel \mathtt{T}^i(E[\textbf{new}\ \mathtt{T}\,(\overline{t}\texttt{;})]), \mathcal{L}, \mathcal{S}\rangle \Longrightarrow \langle Q \parallel \mathtt{T}^i(E[\,]\texttt{;}) \parallel \mathtt{T}^j(\overline{t}\texttt{;}), \mathcal{L}, \mathcal{S}\rangle} \quad \text{[Thread-Fork-R]}$$

$$\frac{\langle t, \mathcal{L}, \mathcal{S}\rangle \longrightarrow \langle t', \mathcal{L}', \mathcal{S}'\rangle}{\langle Q \parallel \mathtt{T}^i(E[t]\texttt{;}), \mathcal{L}, \mathcal{S}\rangle \Longrightarrow \langle Q \parallel \mathtt{T}^i(E[t']\texttt{;}), \mathcal{L}', \mathcal{S}'\rangle} \quad \text{[Congruence-R]}$$

$$\frac{\begin{array}{c} \mathit{match}(\mathcal{S}, l(C), \overline{e}, \mathcal{N}, \Theta) \qquad j\ \mathit{fresh} \\ \mathcal{S}' = \{{}^{\mathcal{S}(l)\setminus\mathcal{N}}/_{\mathcal{S}(l)}\}\mathcal{S} \qquad \mathit{rbody}(\overline{e}, C) = (\overline{\overline{x}\,\overline{x'}}, p, \overline{t}) \\ \langle \Theta\, p, \mathcal{L}, \mathcal{S}\rangle \longrightarrow^* \langle \textbf{new}\ \mathtt{B}\,(\mathit{true}), \mathcal{L}, \mathcal{S}\rangle \end{array}}{\langle Q, \mathcal{L}, \mathcal{S}\rangle \Longrightarrow \langle Q \parallel \mathtt{T}^j(\Theta\,\overline{t}\texttt{;}), \mathcal{L}, \mathcal{S}'\rangle} \quad \text{[React-R]}$$

$$\frac{\mathcal{S}(l) = (e, \overline{v}\overline{v'}) \cdot \mathcal{S}'(l)}{\pi_e \mathcal{S}(l) = (e, \overline{v}\overline{v'}) \cdot \pi_e \mathcal{S}'(l)} \quad \text{[Ev-Proj-Incl]}$$

$$\frac{\mathcal{S}(l) = (e', \overline{v}\overline{v'}) \cdot \mathcal{S}'(l)}{\pi_e \mathcal{S}(l) = \pi_e \mathcal{S}'(l)} \quad \text{[Ev-Proj-Excl]}$$

$$\frac{\mathcal{S}(l) = \mathcal{S}'(l) \cdot (e, \overline{v}\overline{v'}) \cdot \mathcal{S}''(l) \qquad (e, \overline{v}\overline{v'}) \notin \mathcal{S}'(l)}{\mathcal{S}(l) \setminus \{(e, \overline{v}\overline{v'})\} = \mathcal{S}'(l) \cdot \mathcal{S}''(l)} \quad \text{[Ev-Rem]}$$

$$\frac{\overline{s} = \overline{s''} \cdot s'_1 \cdot \overline{s'''} \qquad s'_{2..q} \in \overline{s'''} \qquad P(s'_1, \ldots, s'_q)}{s'_{1..q} \in \overline{s} \qquad P(s'_1, \ldots, s'_q)} \quad \text{[Ev-Seq-Incl]}$$

$$\frac{\begin{array}{c} \forall e_j \in \overline{e} \quad \left(\begin{array}{c} (e_j, \overline{v^j}\ \overline{v'^j})_{1..n_j} \in \pi_{e_j} \mathcal{S}(l(C)) \\ \mathit{etype}(e_j, C) = n_j \times \ldots \end{array}\right) \\ \mathit{rbody}(\overline{e}, C) = (\overline{\overline{x}\,\overline{x'}}, p, \overline{t}) \\ \mathcal{N} = \bigcup_{k \in [1..n_j]} \overline{(e_j, \overline{v^j v'^j})_k} \\ \Theta = \{{}^{l}/\textbf{this}, {}^{\overline{(\overline{v}\overline{v'})_k}}/_{\overline{(\overline{x\,[k-1]}\ \overline{x'\,[k-1]})_{k \in [1..n_j]}}}\} \end{array}}{\mathit{match}(\mathcal{S}, l(C), \overline{e}, \mathcal{N}, \Theta)} \quad \text{[Patt-Match]}$$

Fig. 4. Contextual semantics of FEJ.

3.3 Constraints on Event Methods

Event methods are specific methods, and their declaration and implementation thus follows special restrictions.

R1 Event methods cannot throw exceptions and cannot return values. Their return type is **`event`**. This simplifies broadcast – the absence of exceptions and return values avoids dealing with multiple returns.

R2 Event method headers cannot be **`synchronized`**, as this would contradict their asynchronous nature. Reactions may be defined to be **`synchronized`** though by adding the keyword in front of the correlation pattern declaration.

R3 Similarly, **`final`** applies to correlation patterns. By prefixing a correlation pattern with that keyword, *all* event methods in the correlation pattern are transitively made final, and none of them can be overridden in a subclass correlation pattern.

R4 Predicates, like reactions, can not be defined in interfaces. An interface, or a class, can define an **`abstract`** event correlation pattern, which is strictly the same as defining the respective event methods individually.

R5 A reaction body can make a call to a reaction body of a pattern in its superclass through **`super`** only if the pattern *involves the same set of events*.

R6 An event method can only appear in a single correlation pattern within a class. Without this restriction, semantics become much more complicated, as elaborated in Section 6.

$$\boxed{\langle t, \mathcal{L}, \mathcal{S}\rangle \longrightarrow \langle t', \mathcal{L}', \mathcal{S}'\rangle}$$

$$\frac{\mathcal{L}(l) = [f_1 : v_1, \ldots, f_q : v_q]}{\langle l(A)\,.\,f_j, \mathcal{L}, \mathcal{S}\rangle \longrightarrow \langle v_j, \mathcal{L}, \mathcal{S}\rangle} \quad \text{[Field-Acc-R]}$$

$$\frac{\mathcal{L}(l) = [f_1 : v_1, \ldots, f_q : v_q] \qquad \mathcal{L}' = \{^{[f_1 : v_1, \ldots, f_j : v', \ldots, f_q : v_q]}/_{l}\}\mathcal{L}}{\langle l(C)\,.\,f_j := v', \mathcal{L}, \mathcal{S}\rangle \longrightarrow \langle v', \mathcal{L}', \mathcal{S}\rangle} \quad \text{[Field-Ass-R]}$$

$$\frac{A \preceq T}{\langle (T)\, l(A), \mathcal{L}, \mathcal{S}\rangle \longrightarrow \langle l(A), \mathcal{L}, \mathcal{S}\rangle} \quad \text{[Obj-Cast-R]}$$

$$\frac{D \preceq T}{\langle (T)\, \mathbf{new}\ D\,(d), \mathcal{L}, \mathcal{S}\rangle \longrightarrow \langle \mathbf{new}\ D\,(d), \mathcal{L}, \mathcal{S}\rangle} \quad \text{[Val-Cast-R]}$$

$$\langle \overline{v}\,;\,\mathbf{return}\ v', \mathcal{L}, \mathcal{S}\rangle \longrightarrow \langle v', \mathcal{L}, \mathcal{S}\rangle \quad \text{[Ret-R]}$$

$$\frac{mbody(m, A) = (\overline{x}, \overline{t})}{\langle l(A)\,.\,m\,(\overline{v}), \mathcal{L}, \mathcal{S}\rangle \longrightarrow \langle \{^{\overline{v}}/_{\overline{x}}, {}^{l}/_{\mathbf{this}}\}\overline{t}, \mathcal{L}, \mathcal{S}\rangle} \quad \text{[Meth-R]}$$

$$\langle \mathbf{for}\ i\ \mathbf{in}\,[n\,..\,n']\ p, \mathcal{L}, \mathcal{S}\rangle \longrightarrow \langle \{^{n}/_{i}\}p\ \&\&\ \ldots\ \&\&\ \{^{n'}/_{i}\}p, \mathcal{L}, \mathcal{S}\rangle \quad \text{[Forall-R]}$$

$$\langle !\,\mathbf{new}\ \mathrm{B}\,(b), \mathcal{L}, \mathcal{S}\rangle \longrightarrow \langle \mathbf{new}\ \mathrm{B}\,(\neg b), \mathcal{L}, \mathcal{S}\rangle \quad \text{[Bool-Neg-R]}$$

$$\frac{(b\text{-}op,\ a\text{-}op) \in \{(\&\&, \wedge), (||, \vee)\}}{\langle \mathbf{new}\ \mathrm{B}\,(b)\ \ b\text{-}op\ \mathbf{new}\ \mathrm{B}\,(b'), \mathcal{L}, \mathcal{S}\rangle \longrightarrow \langle \mathbf{new}\ \mathrm{B}\,(b\ \ a\text{-}op\ b'), \mathcal{L}, \mathcal{S}\rangle} \quad \text{[Bool-Op-R]}$$

$$\frac{(v\text{-}op,\ a\text{-}op) \in \{(==, =), (<=, \leq), (<, \leq)\}}{\langle \mathbf{new}\ D\,(d)\ \ v\text{-}op\ \mathbf{new}\ D\,(d'), \mathcal{L}, \mathcal{S}\rangle \longrightarrow \langle \mathbf{new}\ \mathrm{B}\,(d\ \ a\text{-}op\ d'), \mathcal{L}, \mathcal{S}\rangle} \quad \text{[Val-Op-R]}$$

$$\langle l\,.\,e\,(\overline{v}), \mathcal{L}, \mathcal{S}\rangle \longrightarrow \langle l\,.\,e\,(\overline{v})\ [\overline{\nabla t^{l}}], \mathcal{L}, \mathcal{S}\rangle \quad \text{[Ev-Def-R]}$$

$$\frac{\mathcal{S}' = \{^{\mathcal{S}(l)\cdot(e, \overline{v}\ \overline{v'})}/_{l}\}\mathcal{S}}{\langle l\,.\,e\,(\overline{v})\ [\overline{v'}], \mathcal{L}, \mathcal{S}\rangle \longrightarrow \langle \mathbf{new}\ \mathrm{E}\,(), \mathcal{L}, \mathcal{S}'\rangle} \quad \text{[Ev-R]}$$

$$\frac{fields(A) = \overline{T}\ \overline{f} \qquad l \notin dom(\mathcal{L}) \qquad \mathcal{L}' = \{^{[f_1 : v_1, \ldots, f_q : v_q]}/_{l}\}\mathcal{L}}{\langle \mathbf{new}\ A\,(\overline{v}), \mathcal{L}, \mathcal{S}\rangle \longrightarrow \langle l(A), \mathcal{L}', \mathcal{S}\rangle} \quad \text{[Loc-R]}$$

$$\frac{\overline{l} = \{l \mid l(C) \in \mathcal{L} \wedge C \preceq C'\}}{\langle C'\,.\,e\,(\overline{v}), \mathcal{L}, \mathcal{S}\rangle \longrightarrow \langle l_1\,.\,e\,(\overline{v})\ [\overline{\nabla t^{l_1}}]\,;\,\ldots\,;\,l_q\,.\,e\,(\overline{v})\ [\overline{\nabla t^{l_q}}], \mathcal{L}, \mathcal{S}\rangle} \quad \text{[Ev-Bcast-Def-R]}$$

$$\frac{\overline{l} = \{l \mid l(C) \in \mathcal{L} \wedge C \preceq C'\}}{\langle C'\,.\,e\,(\overline{v})\ [\overline{v'}], \mathcal{L}, \mathcal{S}\rangle \longrightarrow \langle l_1\,.\,e\,(\overline{v})\ [\overline{v'}]\,;\,\ldots\,;\,l_q\,.\,e\,(\overline{v})\ [\overline{v'}], \mathcal{L}, \mathcal{S}\rangle} \quad \text{[Ev-Bcast-R]}$$

Fig. 5. Contextual semantics of FEJ (cont'd). $\preceq$ denotes subtyping, refer to the companion technical report [13] for the subtyping rules.

In FEJ, R1 is achieved by the introduction of the placeholder E. R2 and R3 are abstracted, R4 and R6 are enforced by inheriting from FJ. R5 is abstracted in FEJ because FJ does not have **`super`**-calls.

3.4 Event Overloading, Event Overriding and Pattern Overriding

As in Java, an event method e_1 *overloads* e_2 if they have the same name but different type signatures (*etype* in Figure 3). In EventJava, they are treated as two different event methods and can appear in different correlation patterns in the same class (subject to restriction R6). Overriding an event method e (with window $[n]$) is possible (with $[n']$, $n' \geq n$) iff e is not in a **`final`** pattern in the (non-**`final`**) super-class. Consider a correlation pattern p_1 in class C containing event methods $e_1, \ldots, e_q$ with windows $n_1, \ldots, n_q$. Assume that class C' inherits from class C, and defines a pattern p_2 containing e_1. Then, we say that p_2 overrides p_1. But p_2 does not have to contain $e_2, \ldots, e_q$, which may be included in other patterns of C'. So, a pattern in C can be overridden by more than one pattern in C'. By restriction R6, since an event method can occur only in one correlation pattern per class, if C' does not define patterns containing $e_2, \ldots, e_q$, then they become abstract just like the subclass C' itself.

3.5 Global Progress

Consider an object $l(C)$ with a pattern $\overline{\texttt{event}\ e\,[n]\,(\overline{N}\ \overline{x})\,[\overline{N}\ \overline{x}]}\ \texttt{when}\ p\,\{\overline{t};\}$ defined in C.

Definition 1 (Configurations). *We refer to* $\mathcal{C} = \langle Q, \mathcal{L}, \mathcal{S}\rangle$ *as a* configuration.

- $\mathcal{E}_j^k(Q, \mathcal{L}, \mathcal{S}) = \langle \mathrm{T}^i(E[d_j.e_j(\overline{v^j}\ \overline{v'^j})]_k) \parallel Q, \mathcal{L}, \mathcal{S}\rangle$ *is an* event *configuration.*
- $\mathcal{R}_{l(C)}^{\overline{e}}(\Theta, Q, \mathcal{L}, \mathcal{S}) = \langle \mathrm{T}^{\langle l(C), \overline{e}\rangle}(\Theta\overline{t};\ldots) \parallel Q, \mathcal{L}, \mathcal{S}\rangle$ *is a* reaction *configuration.*

Definition 2 (Run).
A run *is a succession of configurations* $\overline{\mathcal{C}}=\mathcal{C}_1 \Longrightarrow \ldots \Longrightarrow \mathcal{C}_q$.

Theorem 1 (Global progress). *Assume a run* $\overline{\mathcal{C}}$ *s.t.* $\forall j \in [1..q], k \in [1..n_j]$, *(i)* $d_j^k \in l(C) \cup \{C' \mid C \preceq C'\}$, *(ii)* $\exists \mathcal{L}_j^k(l)$, *(iii)* $\exists \mathcal{C} = \mathcal{E}_j^k(Q_j^k, \mathcal{L}_j{}^k, \mathcal{S}_j^k) \in \overline{\mathcal{C}}$, *(iv)* $\langle \{{}^{l}/_{\texttt{this}}, \overline{{}^{(\overline{v^j\, v'^j})_k}/_{(\overline{x^j\, x'^j})_{k\in[1..n_j]}}}\}p, \ldots\rangle \longrightarrow^* \langle \texttt{new}\ \mathrm{B}(true), \ldots\rangle$ *. Then* $\forall \overline{\mathcal{C}'} = \overline{\mathcal{C}} \Longrightarrow \overline{\mathcal{C}''}\ \exists\ \mathcal{R}_l^{\overline{e}}(\Theta, Q, \mathcal{L}, \mathcal{S}) \in \overline{\mathcal{C}'}$

Proof by induction on derivation of $\Longrightarrow$. (The theorem reads "If a pattern of an object gets satisfied, the corresponding reaction will eventually be evaluated.")

4 Implementation

This section first presents the implementation framework underlying EventJava. Then, a reference implementation based on Jess [10] and JGroups [11] is presented along with its specific matching semantics, showing that these preserve total ordering properties of message dissemination in JGroups.

4.1 Implementation Framework

The EventJava compiler, implemented using Polyglot [17], translates EventJava programs to standard Java by (a) code transformations and (b) generation of application-specific helper classes (e.g. for broadcasting).

Framework components. The generated code represents the glue between EventJava programs and the framework components shown in Figure 6. An event notification/method invocation is forwarded to the communication *substrate*, JGroups in the case of our reference implementation, which takes care of remote communication including unicast and broadcast. In the broadcast case, the substrate delivers all the serialized event method invocations to the *resolver*, which determines the classes on which the methods were invoked and interacts with *broadcast objects* for those classes. Broadcast objects deliver the events to the sinks, where they are stored, typically but not necessarily, in event queues. The *matcher* — one instance per sink — checks the stored events for a match to any of the correlation patterns and spawns the reaction on its sink. Multithreading can be used in various places, with synchronization depending on the desired semantics. While the substrate and matcher, like the context, are defined as an API, the resolver and broadcast classes are generated by our compiler to avoid costly dynamic invocations through reflection. The context of a given event is used and sometimes modified or augmented throughout the substrate and the matcher.

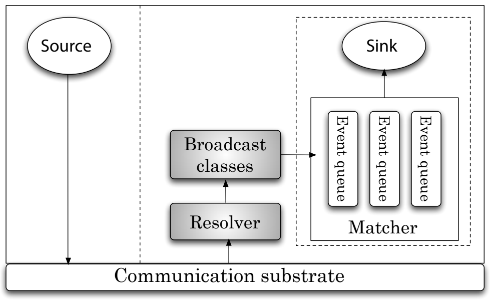

Fig. 6. The EventJava framework. Ovals represent the application. Shaded boxes represent fixed components; others are customizable.

Code transformations. On the source side, each event method invocation is altered to create the context, serialize the explicit arguments, and invoke the substrate. All the instances of any class C which has at least one event method need to be tracked by its broadcast class. To that end, a static field `instances` is added to every such *sink class* to track all of its instances with weak references. Every **`new`** on a sink class is instrumented to add the created object to the class' `instances` set. Broadcast objects for sink classes recursively store references to broadcast objects for their sink subclasses.

Integration with Java RMI and garbage collection of sinks. Some constraints in EventJava come from its integration with the Java RMI framework [18]. This does not mean that remote communication in EventJava takes place over Java RMI. EventJava is merely integrated with the *interfaces*, for portability and interoperability with J2EE. The constraints introduced by this integration lead to a leaner and simpler model and do not reduce expressiveness to the extent of offsetting the benefits of the integration. The integration implies that events must be declared in interfaces subtyping `java.rmi.Remote` (omitted in the examples so far for brevity), which means that sinks are remote objects. Event methods become thereby **`public`**, and can not be **`static`**. These last two restrictions are ensured by FEJ, as bare FJ only supports such members. This integration with Java RMI also helps garbage collection of dead sinks, and ensures that events are not delivered to dead sinks. The **static** field `instances` added to the sink class uses weak references, which are periodically purged.

4.2 Deterministic Matching in the Jess Reference Implementation

While non-determinism might be desired in certain cases, a trading algorithm replicated for reliability by running several instances of the same class will yield contradictory results with the default semantics in Section 3.2, even if the application-level algorithm is deterministic.

Rete-based matching. Figure 7 presents an alternative deterministic dispatching semantics describing our reference implementation of the matcher on top of the Rete [9] algorithm in Jess [10]. In short, Rete treats events, with their explicit and implicit attributes as typed data. The matcher implementation ensures that predicate evaluation is synchronized for a given sink. Correlation patterns and predicates are encoded by our compiler as Jess rules. The matcher delivers the matched events to a dispatch method. The dispatch method, generated by our compiler, has code to receive matched events and use threads from a thread pool to execute the reaction bodies.

Semantics. In Figure 7, rules [Ev-Bcast-Def-R'], [Ev-Bcast-R'] and [Patt-Match'] replace [Ev-Bcast-Def-R], [Ev-Bcast-R] and [Patt-Match] respectively. Rules [React-R'$_1$] and [React-R'$_2$] replace [React-R]. In [Ev-Bcast-Def-R'] and [Ev-Bcast-R'], when an event is broadcast, the context terms are instantiated and the events are added to the corresponding per-object queues of $\mathcal{S}$ in a single atomic step, i.e., total order broadcast is used. This differs from the default semantics of FEJ where a multi-send is used. Again, in [Ev-Bcast-Def-R'], for a given object l, the ∇t_1^l terms assigned must evaluate to values that are totally ordered, in increasing order. This, in combination with the use of total order broadcast, ensures *global* total order, i.e., the events in all the queues of $\mathcal{S}$ are totally ordered.

In Rete-based matching, for pattern **event** e_1`()`,...,e_q`()` **when** p, the *first* received instance of e_1 is chosen for which an instance of each remaining event type has been received such that the predicate p is matched. If there are several instances of e_2 for which instances of $e_3, .., e_q$ exist such that p holds, then the first one is chosen and so on. If an event e_j has an assigned window of size n_j then the algorithm of course looks for the first sequence of length n_j (relation $\in^1$ defined by [Ev-First-Seq-Incl]) such that there are instances of the remaining event types.

Once a match is determined for a given correlation pattern, any event which is of an event type within the correlation pattern and *older* than the respective matching one is discarded in addition to the matching one ($\backslash^*$). Otherwise, the total order determined by JGroups is not preserved. Furthermore, reactions for a same correlation pattern on a same object are executed sequentially in the order in which they are identified, by identifying threads by a $\langle object, pattern\rangle$ tuple ([React-R'$_{1,2}$]). Synchronization code has to consider this. *Total order* broadcast

$$\dfrac{\overline{s}{=}\overline{s'''}\cdot s'_1\cdot\overline{s''''}\quad s'_{2..q}\in^1\overline{s''''}\ P(s'_1,\ldots,s'_q) \\ \nexists\ s''_1\in\overline{s'''}\ :\ \begin{pmatrix}\overline{s}=\ldots\cdot s''_1\cdot\overline{s'''''} & s''_{2..q}\in\overline{s'''''} \\ & P(s''_1,\ldots,s''_q)\end{pmatrix}}{s'_{1..q}\in^1\overline{s}\ P(s'_1,\ldots,s'_q)}\ \text{[Ev-First-Seq-Incl]}$$

$$\dfrac{\mathcal{S}(l){=}\mathcal{S}'(l)\cdot(e,\overline{v}\overline{v'})\cdot\mathcal{S}(l)''\quad (e,\overline{v}\overline{v'})\notin\mathcal{S}'(l)}{\mathcal{S}(l)\setminus^*\{(e,\overline{v}\overline{v'})\}{=}\mathcal{S}(l)''}\ \text{[Ev-Rem-All]}$$

$$\dfrac{\overline{l}{=}\{l\mid l(C)\in\mathcal{L}\wedge C\preceq C'\} \\ \mathcal{S}'{=}\{^{\mathcal{S}(l_1)\cdot(e,\overline{v}\ \overline{\nabla t^{l_1}})}/_{l_1}\ \ldots\ ^{\mathcal{S}(l_q)\cdot(e,\overline{v}\ \overline{\nabla t^{l_q}})}/_{l_q}\}\mathcal{S}}{\langle C'\texttt{.}e\,\texttt{(}\overline{v}\texttt{)}\,,\mathcal{L},\mathcal{S}\rangle\longrightarrow\langle\textbf{new}\ \texttt{E()},\mathcal{L},\mathcal{S}'\rangle}\ \text{[Ev-Bcast-Def-R']}$$

$$\dfrac{\overline{l}{=}\{l\mid l(C)\in\mathcal{L}\wedge C\preceq C'\}\quad \mathcal{S}'{=}\{\overline{^{\mathcal{S}(l)\cdot(e,\overline{v}\ \overline{v'})}/_{\overline{l}}}\}\mathcal{S}}{\langle C'\texttt{.}e\,\texttt{(}\overline{v}\texttt{)}\ \texttt{[}\overline{v'}\texttt{]},\mathcal{L},\mathcal{S}\rangle\longrightarrow\langle\textbf{new}\ \texttt{E()},\mathcal{L},\mathcal{S}'\rangle}\ \text{[Ev-Bcast-R']}$$

$$\dfrac{match^1(\mathcal{S},l(C),\overline{e},\mathcal{N},\Theta)\quad Q{=}Q'\parallel\texttt{T}^{\langle l,\overline{e}\rangle}(\overline{t'}\texttt{;}) \\ rbody(\overline{e},C){=}(\overline{\overline{x}\overline{x'}},p,\overline{t})\quad \mathcal{S}'{=}\{^{\mathcal{S}(l)\setminus^*\mathcal{N}}/_{\mathcal{S}(l)}\}\mathcal{S} \\ \langle\Theta\,p,\mathcal{L},\mathcal{S}\rangle\longrightarrow^*\langle\textbf{new}\ \texttt{B(}true\texttt{)},\mathcal{L},\mathcal{S}\rangle}{\langle Q,\mathcal{L},\mathcal{S}\rangle\Longrightarrow\langle Q'\parallel\texttt{T}^{\langle l,\overline{e}\rangle}(\overline{t}\texttt{;}\Theta\,\overline{t}\texttt{;}),\mathcal{L},\mathcal{S}'\rangle}\ \text{[React-R'}_1\text{]}$$

$$\dfrac{match^1(\mathcal{S},l(C),\overline{e},\mathcal{N},\Theta)\quad Q{\neq}Q'\parallel\texttt{T}^{\langle l,\overline{e}\rangle}(\overline{t'}\texttt{;}) \\ rbody(\overline{e},C){=}(\overline{\overline{x}\overline{x'}},p,\overline{t})\quad \mathcal{S}'{=}\{^{\mathcal{S}(l)\setminus^*\mathcal{N}}/_{\mathcal{S}(l)}\}\mathcal{S} \\ \langle\Theta\,p,\mathcal{L},\mathcal{S}\rangle\longrightarrow^*\langle\textbf{new}\ \texttt{B(}true\texttt{)},\mathcal{L},\mathcal{S}\rangle}{\langle Q,\mathcal{L},\mathcal{S}\rangle\Longrightarrow\langle Q\parallel\texttt{T}^{\langle l,\overline{e}\rangle}(\Theta\,\overline{t}\texttt{;}),\mathcal{L},\mathcal{S}'\rangle}\ \text{[React-R'}_2\text{]}$$

$$\dfrac{\Big((e_j,\overline{v^j}\ \overline{v'^j})_{1..n_j}\in^1\pi_{e_j}\mathcal{S}(l(C))\Big)_{j=1..q} \\ \overline{e}=e_1\ldots e_q\quad etype(e_j,C){=}n_j\times\ldots \\ rbody(\overline{e},C){=}(\overline{\overline{x}\overline{x'}},p,\overline{t}) \\ \mathcal{N}{=}\bigcup_{k\in[1..n_j]}\overline{(e_j,\overline{v^j v'^j})_k} \\ \Theta{=}\{^l/\texttt{this},\ ^{\overline{(\overline{v}\overline{v'})_k}}/_{\overline{(\overline{x\,[k-1]}\ \overline{x'\,[k-1]})}_{k\in[1..n_j]}}\}}{match^1(\mathcal{S},l(C),\overline{e},\mathcal{N},\Theta)}\ \text{[Patt-Match']}$$

Fig. 7. Deterministic matching semantics in the Jess reference implementation.

(as well as reaction serialization) can be disabled in our reference implementation if an application does not require the ordering guarantees. In the absence of ordering guarantees, an EventJava implementation could, for instance, choose to handle reactions like transactions with an optimistic concurrency model.

Ordering properties. We can prove that ordering at the JGroups level is preserved by the matching semantics. Consider Definitions 1 and 2 for configurations and runs given in Section 3.5. Assume $l(C)$ and $l'(C)$, and a pattern $\texttt{event}\ \overline{e\,\texttt{[}n\texttt{]}\,\texttt{(}\overline{N}\ \overline{x}\texttt{)}\,\texttt{[}\overline{N}\ \overline{x}\texttt{]}}\ \texttt{when}\,p\,\texttt{\{}\overline{t}\texttt{;\}}$ defined in C.

Theorem 2 (Order preservation). *Assume a run* $\overline{\mathcal{C}}$ *s.t.* $\forall\mathcal{C}=\mathcal{E}^k_j(Q,\mathcal{L},\mathcal{S})\in\overline{\mathcal{C}}\ d^k_j\notin\{l(C),l'(C)\}$.

Then $\forall\mathcal{C}_i,\mathcal{C}_{i'},\mathcal{C}_j,\mathcal{C}_{j'}\in\overline{\mathcal{C}}\mid\mathcal{C}_i=\mathcal{R}^{\overline{e}}_l(\Theta,Q_i,\mathcal{L}_i,\mathcal{S}_i)$, $\mathcal{C}_{i'}=\mathcal{R}^{\overline{e}}_l(\Theta',Q_{i'},\mathcal{L}_{i'},\mathcal{S}_{i'})$, $\mathcal{C}_j=\mathcal{R}^{\overline{e}}_{l'}(\Theta,Q_j,\mathcal{L}_j,\mathcal{S}_j)$, $\mathcal{C}_{j'}=\mathcal{R}^{\overline{e}}_{l'}(\Theta',Q_{j'},\mathcal{L}_{j'},\mathcal{S}_{j'})$ $i<i'\Leftrightarrow j<j'$.

Proof by induction on derivation of $\Longrightarrow$. (The theorem reads: “For two instances of a same class receiving only broadcast events, the objects will execute reactions to a given pattern in the same order.”)

5 Evaluation

Given that there is a strong variance in workloads produced by distributed applications (same or different), over time depending on their deployment, we do not evaluate our system with specific applications, but rather use stress testing by varying the different parameters of the load. This section stress tests our reference implementation of EventJava (referred to as EventJava) by comparison with (a) the highly tuned Cayuga correlation engine [3] and (b) lightweight limited correlation for concurrency in Cω [22]. All tests use the more resource-demanding "\" semantics (see Section 6).

5.1 Cayuga

Cayuga [3] is a highly tuned, database-backed, correlation engine. The paper by Demers et al. [3] shows that Cayuga outperforms other correlation engines like Aurora [4] and Borealis [5]. All measurement scenarios and settings were taken from [3] to not favor EventJava. Figure 8 compares the throughput of EventJava with Cayuga with respect to the number of different event methods/event types involved per sink.This experiment was conducted on an iMac dual core 2.0Ghz with 2GB RAM. Sink classes were generated with 1000, ... ,150000 (non-abstract) event methods and 4 event methods per correlation pattern, i.e. for the sink class with 100,000 event methods, there were 25,000 correlation patterns. The number of event methods and correlation patterns is relevant because it directly affects the performance of the matcher – the time taken by any search algorithm to match an event to a pattern. The throughput (number of events processed per second) of EventJava remains well above 10,000 events/sec even for the case involving 150,000 event methods and 37,500 correlation patterns, even outperforming Cayuga. Note though that according to [3], Cayuga scales relatively better than EventJava, performance with EventJava drops relatively sharper beyond 150,000 event methods per sink. Cayuga's throughput drops beyond 10,000 event types, but Cayuga can scale even up to 400,000 event types (their throughput is 2000 events/sec at 400,000 event methods). Also, Cayuga's memory footprint is smaller than EventJava. We weren't able to reproduce these results, and use the figures from [3] to plot the graph. Note that only one sink is used because Cayuga has a single correlation engine and the goal is to compare peak throughput of matching. We conclude that even when implemented with custom off-the-shelf components such as Jess, the performance of EventJava is comparable to a highly tuned correlation engine in substantial load scenarios. This illustrates that the high-level programming abstractions of EventJava and its resulting gains in safety, (e.g. when compared to queries expressed in SQL-like grammars) do not entail any inherent penalty.

5.2 Complexity of Correlation Patterns

Figure 9 illustrates the scalability of EventJava with respect to the number of event methods in a correlation pattern. The experiment was conducted using a single sink with 100,000 different event methods. So, if there are 4 event methods per correlation patten, there are 25,000 correlation patterns. The throughput increases slightly with the number of events in the pattern, and in all cases the throughput is well above 14,000 events/sec. Figure 9 shows five such scenarios

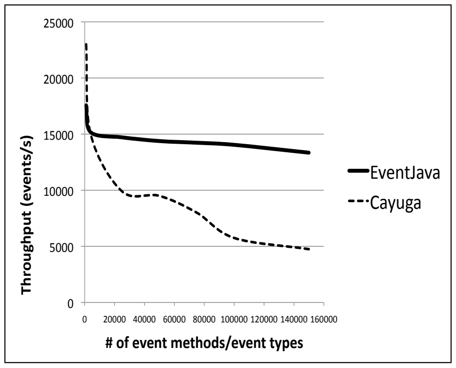

Fig. 8. Simple throughput comparison of EventJava and Cayuga.

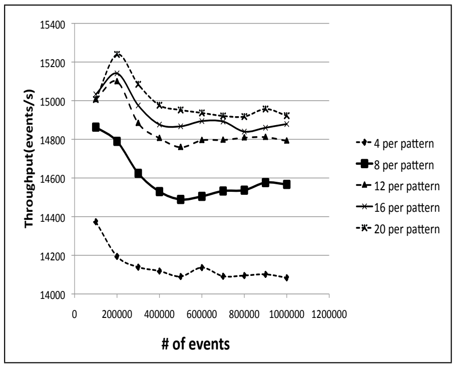

Fig. 9. EventJava throughput w.r.t. number of event methods per pattern.

with 4, 8, 12, 16, and 20 event methods per pattern respectively. In Figure 9, we measure throughput by randomly generating events. The throughput remains fairly constant, irrespective of the number of events used in the measurement. For each scenario, we measure average throughput over streams of 100,000 events to 1 million events. The variation in throughput for any scenario is within 250 events per second, i.e., ∼2%. This shows that the throughput of EventJava does not decrease over time when it faces continuous streams of events. This experiment was conducted on an iMac 2.0 Ghz dual core with 2GB RAM.

5.3 Cω

Polyphonic C# [22], which is now part of Cω, implements the Join calculus [23] in C#. The key differences between (the Polyphonic-C# part of) Cω and EventJava are (i) Cω does not support predicates (ii) Cω targets concurrent programming, supporting one synchronous method per pattern at most (iii) Cω does not explicitly support broadcast interaction (iv) Cω and EventJava differ in the algorithms used for the storage and matching of events, and (v) Cω has *stream types* which can be viewed as pointers to/iterators over a priori endless arrays, but they are not integrated with chords and correlation over streams is not supported. Correlation patterns without predicates are called *chords* in Cω terminology. Calls to asynchronous methods part of a pattern (a *chord* in Cω terminology) are queued, and a reaction can be dispatched when every method in the pattern has been called. In Figure 10, we measure the matching performance of EventJava with Cω for predicate-less patterns which favors the concurrency scenarios aimed at by Cω. The measurements in this case were conducted on an HP PC with an Intel quad core 2.4Ghz processor and 3.5GB RAM.The throughput of EventJava is actually 18-19% higher than that of Cω, which shows the versatility of the reference implementation of EventJava. We conclude that EventJava can be an alternative to Cω for concurrent programming. Note that the introduction of predicates is in fact debated in [22], but not realized to retain the lightweight matching implementation.

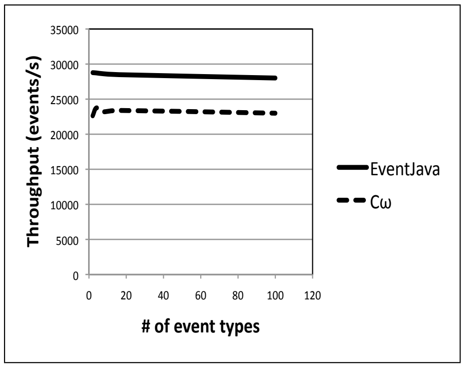

Fig. 10. Simple throughput comparison of EventJava and Cω.

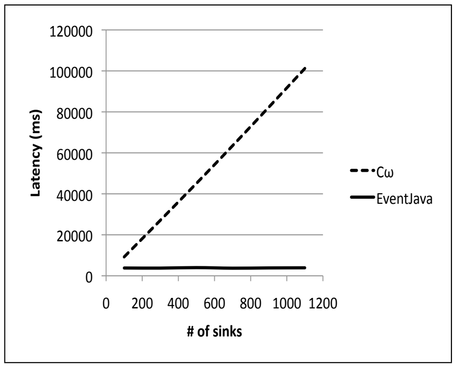

Fig. 11. End-to-end latency of EventJava application with respect to the number of sinks on different nodes.

5.4 EventJava Latency

For completeness, and to argue for the integration of a broadcast substrate in EventJava, we evaluate the end-to-end latency of EventJava in a distributed settting. Latency here is measured as the time interval between the production of the last event that instantiates a correlation pattern, and the dispatch of the corresponding reaction at a possibly remote sink. For example if two events e_1, e_2 are used to match a pattern, and if the reaction at the remote sink is dispatched at time t, then the end-to-end latency is $t - max(e_1\texttt{.time}, e_2\texttt{.time})$. These measurements were conducted in a local area network, where clocks of hosts were closely synchronized. Figure 11 compares the average latency of EventJava with that of the same application implemented using Cω with .NET Remoting. The sink objects were distributed in groups of 100 on 1, 3, 5, 7, 9, 11 nodes and the source was on a different node. Each node was a Dell OptiPlex GX270 Workstation with a 3Ghz Pentium 4 processor and 512 MB RAM running Microsoft Windows XP. Figure 11 shows that average end-to-end latency remains closely constant in the EventJava application as the number of sinks increases, while average latency rapidly increases when performing a blunt multi-send with .NET Remoting.

6 Discussion

We discuss issues related to the design and implementation of EventJava, including three parameters for matching (M1, M2, M3) that can be set by the runtime.

Events in multiple patterns. As mentioned in Section 3.3, in a class, the same event method cannot be a part of more than one correlation pattern. Consider a class C, where an event method e occurs in more than one pattern p_1 and p_2. At runtime, the implementation has two alternatives when an event matches more than one pattern:

A1 *Non-deterministic choice:* Non-deterministically choose a pattern that consumes the event. This breaks the order preservation property of our reference implementation, which would defeat the purpose of many event dissemination protocols in the substrate. An application developer can easily separate p_1 and p_2 into two separate classes C_1 and C_2, and if non-determinism is desired, the developer can easily introduce it by randomization. But, reconstructing event order at the application level is much more complicated. If A1 can also create scenarios where a pattern is *starved*, i.e. does not consume any event for long periods of time.

A2 *Cloning:* Clone the event, thereby allowing all matching patterns to consume the event. This alternative can also break the order preservation property. Also, if events are cloned, the EventJava runtime has to maintain per-pattern data structures to store events, because the consumption of an event by one pattern is independent of other patterns. This degrades performance. A2 also complicates inheritance; if a class C defines two patterns p_1 and p_2, both containing event method e, and if class C' extends C and defines pattern p_3 containing e, does p_3 override p_1 or p_2? Both? Neither? We would need to add further syntax to EventJava to explicitly specify overridden events and patterns.

Because of these drawbacks, EventJava does not permit the same event method to occur in multiple patterns in a class.

Broadcast vs multicast. Through the presence of predicates in EventJava, broadcasting leads to *implicit* multicasting, as not all instances of a sink class C (and of its subsclasses) will necessarily deliver a given event $C.e(...)$. An intermediate case between unicast and implicit multicast consists in an *explicit* multicast where a select set of sinks are addressed – atomically as opposed to a multi-send as portrayed in rule [Ev-Bcast-R] of Figure 5. Several middleware systems propose such protocols natively, or they can be built on top. EventJava supports such interaction through specific proxies. As the invocation then occurs just like a regular unicast invocation (on the proxy) and many authors have elaborated on that in the past (e.g., [19,20]) we omit its presentation.

Bootstrapping and groups. Bootstrapping of EventJava components occurs like in any distributed application: a federated name is necessary for connecting parties. This name defines a group, which delimits an EventJava application and thereby also broadcasts. There are several ways of further reducing the scope of broadcasts. Two dynamic solutions are alluded to above. (1) By adding a *name* attribute to corresponding events, sinks can use predicates to specify *sub*groups of interest. (2) Creating explicit multicast groups by the use of proxies and libraries. Additionally, configuration files can be used to define boundaries for broadcasts on a per-event basis, e.g., through subnet masks.

Order. The ordering property stated above only holds for two instances of a same class, and with respect to individual patterns. By **matching following the patterns of a class in a deterministic order** (M1), which can be enabled in our implementation, the property can be widened to reactions to *all* patterns of two instances of a same class. Given the possibility of redistributing events across patterns and redefining predicates in subclasses, widening to subclasses is not possible straightforwardly. Similarly, *causal* order [21] is a useful property in asynchronous distributed systems devoid of synchronized clocks, e.g. for debugging. It can be inherently achieved with total order broadcast and *local order* [21], which our reference implementation provides, in settings considering *individual* events. As opposed to traditional message-wise delivery, correlation introduces the possibility of several and thus causally ordered events to be delivered *simultaneously* (in fact this is what many patterns are fishing for), but no two events $e_1 < e_2$ ($<$ representing a *happens-before* relation) can be handled by two subsequent reactions r_1 and r_2 in the inverse order, i.e., r_2 will not handle e_1 after r_1 handles e_2.

Event expiry. Predicates add to the possibility of events never being matched – or matchable. Events of a given type can accumulate if events of other types correlated with it are received at lower rates. Our reference implementation thus allows for the **setting of time-outs on events** (M2), by sources (context) and sinks. Furthermore, the deletion of all earlier events of a type when identifying the first one matching its pattern (" $\setminus^*$ " in rules [React-R'$_{1,2}$]) can be similarly disabled, leading to **retaining older non-matched events** (by using " $\setminus$ " instead – M3). It is easy to show that this does not invalidate Theorem 2. However, the order of the *reaction executions*, though still total, can go against the total order determined on *events* by JGroups.

Language design issues. The `Context` class is not a superclass of all classes containing event methods because of ongoing extensions to EventJava where different event methods in a class can have different contexts. Another design choice would be to have an implicit join if the same parameter name is used in two event methods in a correlation pattern, rather than having it represent two variables that have to be disambiguated. Implicit joins are elegant, but programmers may accidentally use the same parameter name where a join is not intended.

7 Related Work

In this section, we present related work on programming language support for event-based programming with emphasis on correlation. An overview of the most closely related languages/frameworks is given in Table 1.

Language	Joins	Type-A predicates	Type-B predicates	Streams	Addressing
ECO [24]	-	✔	-	-	Broadcast
Java$_{PS}$ [25]	-	✔	-	-	Broadcast
Cω [22]	✔	-	-	-	Unicast
Join Java [26]	✔	-	-	-	Unicast
AWED [27]	✔	-	✔	-	Broadcast
CML [28]	✔(staged)	✔	-	-	Broadcast
StreamFlex [29]	✔	-	-	✔	Unicast
StreamIt [30]	✔	-	-	✔	Unicast
Ptolemy [31]	✔(staged)	-	-	-	Unicast
Scala Joins [32]	✔	✔	-	-	Unicast
Scala Actors [33]	✔(staged)	✔	-	-	Unicast
Erlang [34]	✔(staged)	✔	-	-	Unicast
EventJava	✔	✔	✔	✔	Broadcast

Table 1. Overview of inherent event programming features of related programming languages/frameworks. Languages supporting broadcast also have unicast. Type-A and Type-B predicates are described in Section 2.2.

Concurrency. Like Cω [22], Join Java [26] faithfully implements the Join calculus [23] – providing a means to react to correlated asynchronous method invocations, without predicates, broadcast, and customizable matching. Functional languages like CML [28] and Erlang [34] provide powerful support for event-based programming, but do not explicitly support event correlation. In CML, events are essentially reified as function *evaluations* such as reads or writes on channels, which can be combined. 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. This gives the programmer much control over the exact matching semantics, but means implementing partial matching schemes repeatedly. In many cases, more advanced schemes expressed with staged matching can require "re-inserting" an event, which quickly complicates code. CML provides rich libraries with common operators to mitigate the issues above. Actor-based languages like Erlang [34] and Scala Actors [33] similarly support staged event matching. Scala Joins [32] provide Cω-like join patterns, but does not support Type-B predicates and broadcast interaction.

Jeeg [35] is a concurrency extension of Java imposing ordering of method invocations based on patterns described in Linear Temporal Logic (LTL) in a way similar to the routines in many active object approaches, e.g., [36]. Like CML these approaches do however not allow for the atomic reaction to combinations of incoming calls/events. Responders [37] provide a means of writing responsive threads in a state-machine manner, yielding a safe and effective way of arranging event handling code. However, correlation is not supported, and reactions are synchronous to ensure determinism.

Publish/subscribe, streams and aspects. ECO (*events, constraints, objects*) [24] and Java$_{PS}$ [25] extend C++ and Java respectively for publish/subscribe-like distributed programming, i.e., reacting to singleton events.
StreamIt [30] is a dataflow language targeting fine-grained highly parallel stream applications and providing a highly optimizing native compiler (and a Java translator). While StreamIt programs can be parallelized automatically, the language is hardly suited for general purpose applications because of the lack of data types offered and the restricted programming model. Also, there is no support for event correlation or stream correlation. StreamFlex [29] is a Java API for stream processing inspired by StreamIt but providing high-predictability implemented on top of a real-time virtual machine. StreamIt provides `filter`s and `channel`s, leading to a similar programming model as CML. DirectFlow [38] is a domain specific language that simplifies programming information-flow components by hiding the control-flow interactions between them. Again, there is no explicit support for event correlation. AWED (*aspects with explicit distribution*) [27]) is an aspect language supporting the remote monitoring of distributed applications with distributed pointcuts and advice. EventJava can be viewed as AWED turned inside-out: applications are intentionally written to interact with specific events, which is achieved by the means of limited additional syntax. DJcutter [40] extends AspectJ's with remote joinpoints and pointcuts. However, at the runtime level, DJcutter proposes a centralized aspect-server, which constitutes a bottleneck in a large distributed systems; as many others of the others, DJcutter lacks consistency guarantees as a consequence of poor integration with distribution. Ptolemy [31] is an aspect-oriented language with quantified, typed events, but doesn't support correlation – joins can be performed in a staged manner as described earlier.

8 Conclusions and Outlook

We have presented EventJava, a generic language for event-based programming with event correlation. Our implementation framework allows for adaptation to various settings and systems. We are for instance in the process of implementing a lightweight version of EventJava for mobile computing. The notion of context allows us to easily accommodate *context-aware* applications.

We are currently pursuing two further axes of research, centered around matching semantics and the EventJava framework. First, we are devising annotations for flexibly configuring matching semantics on a per-pattern basis. Second, we are investigating the use of domain-specific aspects for context expression and propagation and other parts of our framework.

9 Acknowledgements

We would like to thank Ryan Maus at Allston Trading LLC for input on the algorithmic trading examples and Jacob Fancher for his contribution to the EventJava compiler. We would also like to thank William Cook, as well as the anonymous reviewers for their invaluable feedback, which helped improve the contents and presentation of this paper.

References

1. Trigeo: TriGeo Security Information Manager (Trigeo SIM). (2007) http://www.trigeo.com/products/detailedf/.
2. Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., Culler, D.: The *nesC* Language: A Holistic Approach to Networked Embedded Systems. In: PLDI 2003. 1–11.
3. Demers, A., Gehrke, J., Hong, M., Riedewald, M., White, W.: Towards Expressive Publish/Subscribe Systems. In: EDBT 2006. 627–644.
4. Abadi, D.J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., Zdonik, S.: Aurora: A New Model and Architecture for Data Stream Management. VLDB Journal **12**(2) (2003) 120–139.
5. Ahmad, Y., Berg, B., Çetintemel, U., Humphrey, M., Hwang, J.H., Jhingran, A., Maskey, A., Papaemmanouil, O., Rasin, A., Tatbul, N., Xing, W., Xing, Y., Zdonik, S.: Distributed Operation in the Borealis Stream Processing Engine. In: SIGMOD 2005. 882–884.
6. Amir, Y., Danilov, C., Miskin-Amir, M., Schultz, J., Stanton, J.: The Spread Toolkit. http://www.spread.org.
7. Pietzuch, P.R., Bacon, J.: Hermes: A Distributed Event-Based Middleware Architecture. In: ICDCSW 2002. 611–618.
8. Apache: ActiveMQ. (2008) http://activemq.apache.org/.
9. Forgy, C.: Rete: A Fast Algorithm for the Many Patterns/Many Objects Match Problem. Artificial Intelligence **19**(1) (1982) 17–37.
10. Friedman-Hill, E.: Jess, http://www.jessrules.com/jess/. (2008)
11. Ban, B.: JGroups - A Toolkit for Reliable Multicast Communication, http://www.jgroups.org/javagroupsnew/docs/index.html. (2007)
12. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: A Minimal Core Calculus for Java and GJ. TOPLAS **23**(3) (2001) 396–450.
13. Eugster, P., Jayaram, K.R.: 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)
14. Chakravarthy, S., Krishnaprasad, V., Anwar, E., Kim, S.K.: Composite Events for Active Databases: Semantics, Contexts and Detection. In: VLDB 1994. 606–617.
15. Sanchez, C., Slanina, M., Sipma, H., Manna, Z.: Expressive Completeness of an Event-Pattern Reactive Programming Language. In: FORTE 2005. 529–532.

16. Welc, A., Hosking, A.L., Jagannathan, S.: Transparently Reconciling Transactions with Locking for Java Synchronization. In: ECOOP 2006. 148–173.
17. Nystrom, N., Clarkson, M.R., Myers, A.C.: Polyglot: An Extensible Compiler Framework for Java. In: CC 2003. 138–152.
18. Sun: Java Remote Method Invocation (Java RMI). (2004) http://java.sun.com/j2se/1.5.0/docs/guide/rmi/.
19. Black, A., Immel, M.: Encapsulating Plurality. In: ECOOP 1993. 56–79.
20. Guerraoui, R., Garbinato, B., Mazouni, K.: GARF: A Tool for Programming Reliable Distributed Applications. Concurrency **5**(4) (1997) 29–32.
21. Toinard, G.F.C.: A New Way to Design Causally and Totally Ordered Multicast Protocols. OSR **26**(4) (1992) 77–83.
22. Benton, N., Cardelli, L., Fournet, C.: Modern Concurrency Abstractions for C#. TOPLAS **26**(5) (2004) 769–804.
23. Fournet, C., Gonthier, C.: The Reflexive Chemical Abstract Machine and the Join Calculus. In: POPL 1996. 372–385.
24. Haahr, M., Meier, R., Nixon, P., Cahill, V., Jul, E.: Filtering and Scalability in the ECO Distributed Event Model. In: PDSE 2000. 83–92.
25. Eugster, P.: Type-based Publish/Subscribe: Concepts and Experiences. TOPLAS **29**(1) (2007)
26. Itzstein, S.V., Kearney, D.: The Expression of Common Concurrency Patterns in Join Java. In: PDPTA 2004. 1021–1025.
27. Navarro, L., Südholt, M., Vanderperren, W., Fraine, B.D., Suvée, D.: Explicitly Distributed AOP using AWED. In: AOSD 2006. 51–62.
28. Reppy, J.H., Xiao, Y.: Specialization of CML Message-passing Primitives. In: POPL 2007. 315–326.
29. Spring, J., Privat, J., Guerraoui, R., Vitek, J.: StreamFlex: High-throughput Stream Programming in Java. In: OOPSLA 2007. 211–228.
30. Lamb, A.A., Thies, W., Amarasinghe, S.: Linear Analysis and Optimization of Stream Programs. In: PLDI 2003. 12–25.
31. Rajan, H., Leavens, G.T.: Ptolemy: A Language with Quantified, Typed Events. In: ECOOP 2008. 155–179.
32. Haller, P., Van Cutsem, T.: Implementing Joins using Extensible Pattern Matching. In: COORDINATION 2008. 135–152.
33. Haller, P., Odersky, M.: Actors that Unify Threads and Events. In: COORDINATION 2007. 171–190.
34. Ericsson Computer Science Laboratory: The Erlang Pogramming Language. www.erlang.org.
35. Milicia, G., Sassone, V.: Jeeg: Temporal Constraints for the Synchronization of Concurrent Objects. CCPE **17**(5–6) (2005) 539–572.
36. Briot, J.P.: Actalk: A Testbed for Classifying and Designing Actor Languages in the Smalltalk-80 Environment. In: ECOOP 1989. 109–129.
37. Chin, B., Millstein, T.: Responders: Language Support for Interactive Applications. In: ECOOP 2006. 255–278.
38. Lin, C., Black, A.P.: DirectFlow: A Domain-Specific Language for Information-Flow Systems. In: ECOOP 2007. 299–322.
39. Bierman, G., Meijer, E., Schulte, W.: The Essence of Data Access in Cω. In: ECOOP 2005. 287–311.
40. Nishizawa, M.: Remote Pointcut: A Language Construct for Distributed AOP. In: AOSD 2004. 7–15.