Program Analysis for Event-based Distributed Systems [*]

K. R. Jayaram
Department of Computer Science
Purdue University
jayaram@purdue.edu

Patrick Eugster
Department of Computer Science
Purdue University
peugster@purdue.edu

ABSTRACT

Designing distributed applications around the idiom of events has several benefits including extensibility and scalability. To improve conciseness, safety, and efficiency of corresponding programs, several authors have recently proposed programming languages or language extensions with support for event-based programming.

The presence of a dedicated programming language and compilation process offers avenues for program analyses to further improve simplicity, safety, and expressiveness of distributed event-based software. This paper presents three program analyses specifically designed for event-based programs: *immutability analysis* avoids costly cloning of events in the presence of co-located handlers for same events; *guard analysis* allows for simple yet expressive subscriptions which can be further simplified and handled efficiently; *causality analysis* determines causal dependencies among events which are related, allowing unrelated events to be transferred independently for efficiency. We convey the benefits of our approach by empirically evaluating their performance benefits.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed Programming*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program Analysis*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed Applications*

General Terms

Languages, Performance

Keywords

event, distributed, correlation, language, program analysis

[*]This research is supported, in part, by the National Science Foundation (NSF) under grants #0644013 and #0834529, and by DARPA under grant #N11AP20014.

1. INTRODUCTION

Event-based design *decouples* system components thus improving extensibility and scalability of the software. More precisely, decoupling is achieved by using a dedicated runtime system that delivers events from *sources* (producers, publishers) to *sinks* (consumers, subscribers) which are not a priori aware of each other, yielding the following benefits:

Simplicity. Decoupling avoids name binding of components, yielding modules which are largely independent. Components can thus be developed, extended, or tested independently from each other. By the same token, new components can be deployed into a running application.

Efficiency. The use of a dedicated middleware system allows for communication to be performed efficiently and in a scalable manner by aggregating and sharing traffic among components where possible.

Event-based middleware systems include *message queues* (an event is consumed by one *of* many sinks) and *publish/subscribe* systems (an event is delivered to *all* sinks interested in it). Systems include research engines (e.g., Siena [10], JEDI [12], and PADRES [28]) or industrial solutions (e.g., Amazon's Simple Queue Service, ActiveMQ, or FioranoMQ), several of which provide both types of communication.

Programming solutions for event-based programming range from simple design patterns (observer design pattern) to language extensions capturing concurrency through asynchronous events (e.g., Ptolemy [35], Responders [11], Scala [20] following an Actor model) to even higher-level coordination and event correlation abstractions (e.g., SCHOOL [32], Cω[7] based on Join Calculus [16]). Language-based approaches have several well-known benefits [9]. First, by providing specific constructs and abstractions these approaches further improve the *simplicity* of programs. Second — and among the prime motivations for language design research in general — choosing the right abstraction can improve the *efficiency* of language implementations. Third, based on language constructs, the compiler can ensure static or dynamic *safety* properties, such as type conformance.

Program analysis offers many intriguing avenues to increase the efficiency and other aspects of distributed event-based applications, but remains underexploited. This is precisely the motivation for this paper. We present three program analyses yielding benefits in terms of simplicity, efficiency, as well as safety:

Immutability analysis: When an event is consumed multiple times by different handlers in a same process then the event and its attributes (representation) must be cloned in most cases to not hamper the decoupling nature of event-based interaction and pass-by-value semantics that are used to support it. Immutability analysis infers cases where attributes are not modified in any way (e.g., by assignment) by handlers or further down the line, such that expensive cloning can be avoided. This may have a cumulative effect as co-locating handlers of a same event allows for limiting the times an event is sent over the wire.

Guard analysis: Guards are the natural way to express subscriptions in event-based languages. Besides allowing for statically verifying compliance of subscriptions with the events they are expressed on, guards naturally capture advanced features such as *parametric subscriptions* [22, 25] which have been shown to benefit efficiency. In contrast to an API-based support for the subscriber-local variables underlying such subscriptions, programming language support allows for complex subscriptions, e.g., those involving arithmetic and multiple variables to be expressed, and for the compiler to automatically simplify these and infer the relevant parameters.

Causality analysis: Causal order between events, especially when these are broadcast, has traditionally been achieved by manually mapping events to "broadcast groups", implemented by means of appropriate protocols [37], to capture dependencies. This is not only tedious for programmers (groups may overlap) but unsafe as dependencies are easily overlooked. Pessimistically funneling all events — even those requiring no ordering with respect to others and each other — through a same group strongly hampers efficiency. Causality analysis infers all *possible* dependencies (for safety) from programs, and with hints from the programmer on independencies (for efficiency) allows for groups to be be created adequately and manipulated automatically by the runtime.

These analyses can implemented on top of event processing middleware API as well with some minor additional effort. For the sake of presentation simplicity, we present them however in the context of our EventJava language [15], which has been designed with emphasis on *genericity*, *extensibility* and *flexibility* to thwart the potential limitations of domain-specific programming languages. EventJava is an extension of the mainstream Java programming language, implemented with an extensible compiler. EventJava's runtime is implemented as a framework of substitutable components, accessible directly through API, which allows existing middleware systems to be plugged in.

In summary, this paper makes the following contributions:

1. We introduce a subset of (mostly) static program analyses tailored to event-based programming which we have devised for EventJava, namely *immutability analysis*, *guard analysis*, and *causality analysis.*

2. We illustrate the benefits of our approach through empirical evaluation of performance gains enabled by our analyses.

Roadmap. Section 2 introduces the EventJava framework. Sections 3-5 present our analyses. Section 6 demonstrates the performance benefits of our approach. Section 7 summarizes related work. Section 8 concludes with final remarks.

2. EVENTJAVA LANGUAGE FRAMEWORK

We present an overview of EventJava. EventJava combines (1) a design leveraging *core abstractions* for fundamental constituents of distributed event-based programming — event *representation*, *production*, *delineation*, *selection*, and *consumption* — and (2) an *open framework*-based implementation. For brevity, we focus on these core abstractions and omit other features or options.

2.1 Event Representation

An application event *type* is implicitly defined by declaring an *event method*, a special kind of asynchronous instance-level method. The formal arguments of an event method correspond to the (explicit) attributes of the event type. For example, quote(**long** time, String org, **float** price) represents the signature of stock quotes. An event method declaration is preceded by the **event** keyword, which makes the distinction between a regular, synchronous, method with **void** return type and an asynchronous event method. This is similar to *chorded* languages (e.g. signal in Join Java [23], async in Cω [7]; cf. Section 7). For instance, an interface Stock could simply declare a quote event:

```
interface Stock ... {
  event quote(long time, String org, float price);
}
```

The knowledge of event *types* in contrast to *structural conformance* [31] has several immediate performance benefits, as events of different types can be handled in parallel. On the other hand, static typing of events does not preclude the addition of new event types at runtime [3]. The small overhead incurred by such infrequent additions is largely outweighed by the gain on every event [24].

2.2 Event Production

Most languages and systems focus on either notifying events to individual parties (unicast) *or* to several parties (multicast). EventJava distinguishes these choices syntactically allowing for "generalized specialization".

2.2.1 Unicast (*one*-to-*one*)

In the simplest case, an event can be notified to a single object by invoking an event method on a single object. For example, a stock quote event can be notified to an instance s of Stock simply as s.quote(...). An event method invocation however decouples the invoker from any invokee (for any production mode) in time. Chorded languages focusing on concurrency mostly follow this unicast addressing scheme.

2.2.2 Multicast (*one*-to-*many*)

The alternative to unicast is multicast, corresponding to publish/subscribe-based systems and languages. For multicast we can distinguish several sub-cases:

a. all-*of*-many: In this case, all objects in an application which implement the event method will be notified. This is achieved by reusing the notation known from **static** methods: Stock.quote(...) dispatches the event to all objects conforming to Stock (including by subtype subsumption). The same kind of call can be made on any class C implementing Stock, limiting the event to all instances of C and its sub-classes. Note that "all" refers to the set of *addressed* objects. All-of-many represents a *potential* broadcast, as the delivery of the event to a particular object — with any production model chosen — will always be subject to any *guards* on that object as we will see shortly.

b. some-*of*-many: This second case corresponds to the explicit addressing of a *set* of objects, or in other terms, to an *explicit* multicast or *group* broadcast. Here, as in group communication scenarios, a group proxy can be used and specific libraries can be offered to select among different protocols for dissemination and membership management upon proxy instantiation.

c. one-*of*-many: As a special case of b. above, one-of-one ensures that an event is not delivered by more than one object. It can be achieved by means of specific proxies. This scenario is referred to as *point-to-point* in Java Message Specification parlance [40].

2.3 Event Delineation

By enabling reactions to *complex* events rather than only individual events, application components can be simplified and repetitive or spurious coordination, composition, and communication can be further avoided. We also refer to such complex events also simply as *patterns*. EventJava enables the delineation of such patterns in *time* and *space*.

2.3.1 Composition in time

Composition in time is supported in EventJava through event *windows* which are syntactically unified with arrays. As an example, we can declare a class Broker which composes streams of events over a window size of 4 as follows:

```
class Broker implements Stock ... {
  event quote[4](long time, String org, float price) {...}
}
```

The attributes of an individual event can be referred to by indexing. For example, quote[2].time represents the time value of the third instance of quote (indices start at 0 just like arrays). This syntax supports efficient implementations by making the number of instances explicit as opposed to other approaches which represent streams as specific types which functions can iterate over by fetching the "next" instance [8]. The syntax provides the best of a declarative correlation style which accounts for the popularity of SQL-derived languages for correlation, in a core imperative model. The time attribute here refers to physical time, and the assumption in this example is that the clocks of producers are synchronized. One therefore expects time to be monotonically increasing with an increasing index i for quote[i]. EventJava includes domain-specific aspects which allow event timestamps to be abstracted as a first-class *context* together with other *implicit* attributes, e.g., source information, credentials. Domain-specific aspects can be used to support other notions of time like vector clocks, when clocks of producers are not synchronized. These are discussed in detail in [21].

2.3.2 Composition in space

Composition in space refers to the ability of composing events of different types through *joins*. These are expressed by comma-separated lists of event method *headers*. For instance, a class Broker2 can combine quote with analyst forecast events as follows:

```
class Broker2 implements Stock ... {
  event quote[4](...), forecast(...) {...}
}
```

The method body, referred to simply as *reaction*, is thus "shared" among the different event method headers. Of course, the separation of time and space for composition is only an abstraction; events composed in space are not all generated at the exact same point in time. In practice composition occurs in a mixed time&space form. Overloading and overriding become more intricate in the presence of composition [15]. In this context, it is sufficient to know that EventJava does not allow an event method to appear in multiple patterns of same class.

2.4 Event Selection

EventJava separates the expression of *which* events are composed from *how* they are composed by the introduction of *guards*. We can extend the example above as follows:

```
class Broker3 implements Stock ... {
  event quote[4](long time, String org, float price),
    forecast(long time, String org, float price)
  when (forecast.price > quote[0].price &&
        quote[0].time > forecast.time &&
        forall k in [0..2] quote[k].time < quote[k+1].time &&
      forall i in [0..2] quote[i].price > quote[i+1].price &&
      forall j in [0..3] quote[j].org == forecast.org) {...}
}
```

to express a strategy consisting in reacting upon a four-fold decrease in the price of a stock following an analyst forecast (assuming synchronized clocks). A guard can use regular Java operators for boolean expressions, such as negation (!) or disjunction (||). The absence of a guard is interpreted as **when true**. Fully qualified notation (e.g., forecast.org) can be simplified by renaming arguments (e.g., using org1 in forecast). Compilation will yield an error message if ambiguity exists. There is no *implicit* matching on homonymous attributes across events. As we will elaborate on more in the context of our analyses, EventJava supports certain uses of local program variables within guards. These allow for *dynamic* subscriptions, preserving the potential for runtime adaptation of late binding of subscriptions (by expression through strings) while avoiding malformed subscriptions.

2.5 Event Consumption

Our design supports two different models of event consumption. The default is asynchronous consumption, which is typical with declarative event correlation patterns – namely that a reaction is triggered *as soon as* a matching set of events has been identified. In the Broker3 example above, asynchronous consumption entails that the reaction is triggered as soon as four quote events and one forecast event *matching* the guard are received. As an alternative to asynchronous consumption, EventJava provides the **queue** keyword as a "substitute" for **event**. More precisely, a comma-separated list of event methods can be preceded by **queue** instead of **event**, with otherwise identical syntax, meaning that matching of the events in the corresponding composition will only be triggered *explicitly* by using the **next** keyword at the beginning of a statement followed by the names of involved events (without attributes). A statement **next** quote, forecast; in any of the methods of Broker3 or its subclasses would trigger the matching. This allows for the separation of complex events and reactions, and their timing.

Alike chorded languages, EventJava supports as syntactic sugar the possibility of including at most one regular, synchronous, method into a complex event. A reaction for a complex event with return type can return a value to the corresponding method invocation via the **return** statement.

2.6 Implementation Framework

The EventJava compiler translates EventJava to standard Java, including calls to interfaces of a runtime framework with substitutable components (see Figure 1). In short, an event notification is passed to the communication *substrate* which takes care of remote communication including unicast and multicast. In the multicast case, the substrate delivers all the serialized event method invocations to the *dispatcher*, which determines the classes on which the methods were invoked and interacts with *multicast objects* (omitted for simplicity) for those classes. These objects pass the events to the sinks, from where they are passed to the *matcher* where they are typically added to an *event store* (e.g., a queue). The matcher is responsible for checking the stored events for completed patterns. The matcher may also apply a garbage collection policy or update/replace stored events.

Serializer, dispatcher and multicast objects represent type-specific code generated at compilation to avoid costly calls through Java reflection. The substrate, matcher, and handler components are defined as APIs. Multicast invocations typically lead to calls to a multicast method of the Substrate interface. The compiler also generates string-based subscription filters from guards. These follow a syntax extending that of *selectors* in JMS [40]. They can be used by a substrate to perform message filtering during propagation [10].

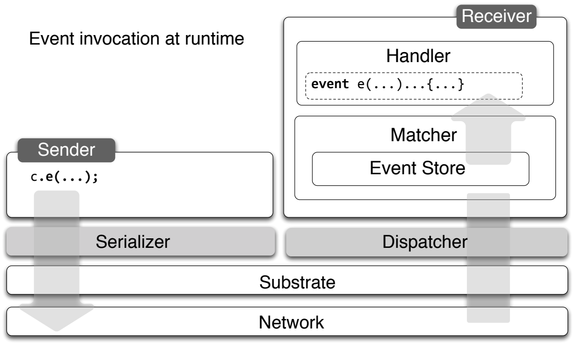


Figure 1: The EventJava runtime framework. Shaded portions represent application-specific components generated by the compiler.

3. IMMUTABILITY ANALYSIS

Immutability analysis avoids costly cloning of events in the presence of co-located handlers for the same event. Immutability analysis is the simplest of the proposed analyses and thus we present it first. It is related to different *pointer analyses* in programming languages.

3.1 Event Attribute Sharing

Like other systems, EventJava accepts Serializable types as event attributes and not only primitive types. While multicasting, such an attribute can be passed to several objects even in a same JVM, and such attributes can be used in reactions just like formal arguments of any method. Since serializability does not imply immutability, and we do not want to restrict the programmer to certain types yet want to avoid unwanted sharing of event attributes, attributes passed to multiple objects would have to be cloned. Yet cloning of objects can become expensive especially if the event attributes being cloned are complex (containing multiple fields and/or several levels of nesting), represent larger structures such as XML structures or image files, or have to be notified to a large number of co-located sinks. Consider, for simplicity below, the slight alteration of the stock trading example

```
class Broker4 ... {
  event ticker[4](long time, Quote q),
    forecast(long time, String org, float price)
  when (forecast.price > ticker[0].q.price &&
    forall i in [0..2] ticker[i].q.price > ticker[i+1].q.price
    && forall k in [0..2] ticker[k].time < ticker[k+1].time
    && ticker[0].time > forecast.time &&
    forall j in [0..3] ticker[j].q.org == forecast.org) {...}
}
```

where Quote is straightforwardly defined as follows:

```
class Quote implements Serializable {
  String org;
  float price;
  ...
}
```

We can have multiple instances of Broker4 in a same address space, or also co-locate these with instances of subclasses of Broker4 which for instance override the guard to implement other strategies. Unless we clone a *quote*'s q attribute for these different reactions/reacting objects, there can be implicit sharing of q.

The body of the reaction depicted in class Broker4, or in a subclass of it, could decide to *retain* certain Quotes to keep track of trends for later analysis or as integral part of correlation strategies. Then, nothing prevents a reaction from modifying any of the fields of q (e.g., q.price =q.price + provision) before say storing q in a collection, while another reaction stores the original q.

3.2 Pessimistic Static Analysis

To avoid unnecessary cloning, yet remain on the safe side to avoid "data races" through the manipulation of event attributes in reactions, we perform an *immutability analysis* on event method arguments. This analysis determines for serializable formal arguments whether they *might be* eventually modified through method invocations or direct assignment to any of their fields. If the analysis can not assert immutability (without inspecting the whole program), the EventJava compiler inserts code to clone matched events before they are used to dispatch the reaction.

4. GUARD ANALYSIS

Next we present the analysis of guards performed in EventJava. In fact this analysis consists in several sub-analyses which are aligned with the guard syntax.

4.1 Dynamic guards

Algorithmic stock trading — one of the classic showcases for publish/subscribe interaction — thrives on rapid adaptations in subscriptions, consisting mostly in adapting the range of permissible values for event attributes [39].

EventJava thus supports *parametric subscriptions* [22, 25], i.e, *dynamic* guards, by allowing for fields of consumer objects to be used in guards. In EventJava an example like the following looks like quite natural:

```
class Broker5 implements Stock ... {
  float targetPrice = ...;
  float thresholdFactor
  event quote(long time, String org, float price)
  when price < thresholdFactor*targetPrice {...}
}
```

Observe that thresholdFactor and targetPrice are not **final** and can thus change throughout the life-time of a Broker4 instance. Any updates to the local variables will take effect.

More precisely, EventJava allows fields of sink classes to be used in guards. By abuse of language but for brevity, we use the term *guard fields* to refer to such fields f (`this.`f). To exploit these fields, assignments have to be supported. This is achieved by a *guard analysis*, statically verifying restrictions on guard fields to ensure that updates to those fields *can be* tracked (without whole-program analysis), and inserting code for tracking actual updates. Of course fully exploiting them requires a corresponding middleware system, which registers for corresponding updates with the framework thus receiving selectors in an extended syntax including variable names. Without this, the updates are translated to issuing a new subscription and canceling the outdated one. As we will illustrate later, support for dynamic guards as enabled by our static analysis improves performance under subscription updates. Note that EventJava allows the evaluation of guard fields to be kept local, if desired, by "pinning" these fields through a modifier local in their declarations.

4.2 Completeness

Tracking all possible updates that affect fields is impossible in practice in the presence of nested types, recursion, and aliasing. EventJava thus imposes the following restrictions (enforced by the type system) on fields used in guards:

R1 Guard fields must be of primitive types.

R2 Guard fields must be **private** or **protected** fields.

These restrictions capture the two possible ways in which fields can be modified, namely by method invocations and assignments. More precisely, R1 ensures that when an object gains access to a field of a sink — for example such a field can be used as return value of a method — no method calls can be performed on the referenced object. Methods can namely have side-effects which could alter an invoked object. Tracking and instrumenting all possible places where method invocations could lead to such alterations can potentially require any class to be instrumented. Thanks to R2, the places where field assignments must be tracked are limited to sink classes and their subclasses.

4.3 Tracking Updates in Single-Field Guards

In this subsection, we consider dynamic guards where event attributes are compared to a single guard field. Even in single-field guards, to track updates, assignments to guards fields are instrumented such as to notify updates to the runtime infrastructure. Since guard fields can appear in several guards for a given sink class, several update notifications might be generated by a single field assignment. In practice these can be combined into single notifications.

It is important to consider the possibility of concurrent updates to guard fields. To ensure that the order of updates is not permuted and that no relevant updates are lost, assignments to guard fields have to occur in mutual exclusion. Any sink class is thus added a *lock field* fLock of type FIFOMutex for each of its guard fields f, to serve as a FIFO mutex, regrouping and protecting updates to f and engendered update(...) calls to the runtime infrastructure (substrate). Such calls in the case of the EventJava runtime infrastructure are made wait-free in that they simply "drop" the update into a buffer, which stores only the last yet unhandled update request for a given parameter. The default implementation for middleware systems which do not inherently support parametric subscriptions leads to a re-subscription – issuing a new subscription and unsubscribing the outdated one.

Instrumentation occurs on every **private** field used in a guard, and on every **protected** field declared by a sink class. The possibility of inheritance with protected fields makes it necessary to pessimistically instrument every modification of such fields declared by sink classes, even for fields which are never used in any guard – not even in respective subclasses. Inversely, if a sink class C has a super-class C' (possibly recursively) which is not subscribed to any events, then C' must be re-compiled if C uses any protected fields of C' in its guards.

4.4 Expressiveness

We would like guards to be as expressive as possible, without entailing much support in the substrate. We discuss extensions to the subscription grammar implemented by EventJava – an extension handled purely by program rewriting and then two extensions benefitting from runtime support.

4.4.1 Parameter expressions

Consider a navigation system that uses GPS sensors to display traffic density around the current location (i.e., GPS X and Y coordinates) of an automobile. This can be expressed in EventJava as follows (the fields constituting subscription parameters are italicized and underlined:

```
class TrafficMonitor {
  float myXPos, myYPos, myXRange, myYRange;
  TrafficMonitor(...) {... /* Init values */ }
  // Subscribe to events from (X, Y) s.t.
  // abs(myXPos - X) ≤ myXRange and abs(myYPos - Y)
       ≤ myYRange
  event trafficDensity(float vehiclesPerSec, float xPos,
       float yPos)
    when (xPos >= myXPos − myXRange &&
          xPos <= myXPos + myXRange &&
          yPos >= myYPos − myYRange &&
          yPos <= myYPos + myYRange) {
          ... // E.g., update navigation screen
  }
  public void setXPos(float newXPos) {
    myXPos = newXPos;
  }
}
```

Our earlier work [25] on parametric subscriptions does not directly support such *complex* expressions on variables

```
class TrafficMonitor{
    float myXPos, myYPos;
    float myXRange, myYRange;
    //Generated Variables
    float _gen1, _gen2, _gen3, _gen4;
    ...
    TrafficMonitor(...) {
      ... // Init values like myXPos, myYPos, myXRange...
      // Instrument the constructor accordingly
      _gen1 = myXPos − myXRange;
      _gen2 = myXPos + myXRange;
      _gen3 = myYPos − myYRange;
      _gen4 = myYPos − myYRange;
    }
    event trafficDensity(float vehiclesPerSec, float xPos,
          float yPos)
      when (xPos >= _gen1 && xPos <= _gen2 &&
          yPos >= _gen3 && yPos <= _gen4 ) {
          // E.g., update navigation system
    }
    public void setXPos(float newXPos) {
      myXPos = newXPos;
      // Instrument methods to reevaluate any generated
            variable
      // that could be affected by an assignment
      _gen1 = myXPos − myXRange;
      _gen2 = myXPos + myXRange;
      //Instrument methods to send update messages
      Substrate.update("_gen1", _gen1);
      Substrate.update("_gen2", _gen2);
    }
}
```

Figure 2: Translating expressions in guards to use single variables. Backlit portions are compiler-generated; *Emphasized* variables represent actual subscription parameters. Their use in guards is underlined.

in a subscription. EventJava can however easily deal with them by translation. To implement parametric subscriptions with expressions, i.e., comparing an event attribute to an expression containing possibly multiple constants and guard fields, the EventJava compiler simplifies these expressions by introducing *virtual* variables into the program which represent the values of high-level expressions. For example, the subscription in the TrafficMonitor class described above is translated as outlined in Figure 2. Backlit portions are compiler-generated. In this figure, *italicized* variables represent *actual* subscription parameters. Their use in guards is underlined. Locks are omitted for presentation simplicity.

In the TrafficMonitor example, each expression in the subscription is captured by a variable that the compiler introduces, e.g., _gen1 captures the value of the combined expression myXPos − myXRange. The compiler also generates code to initialize _gen1 after each of its components, i.e., myXPos and myXRange is initialized in constructors. Also, whenever the value of any component changes (as in setXPos(...)), the values of all generated variables depending on that component (e.g., myXPos) are recomputed and parameter update requests issued.

4.4.2 Attribute expressions

If a traffic monitor were interested in traffic density in a *circular* area of radius myRange, we would like to express this subscription in EventJava as outlined below:

```
class TrafficMonitor2 {
  float myXPos, myYPos;
  float myRange;
  ...
  // Subscribe to events from (X, Y) s.t.
  // ((myXPos - X)^2 + (myYPos - Y)^2)^(1/2) <= myRange
  event trafficDensity(float vehiclesPerSec, float xPos,
        float yPos)
    when (EuclideanDistance(myXPos − xPos, myYPos −
          yPos) <= myRange) {
        ... // E.g., update navigation screen
  }

  static float EuclidianDistance(float xDist, float yDist) {
    return Math.sqrt((xDist ∗ xDist) + (yDist ∗ yDist));
  }
}
```

This example illustrates several things. First, the method EuclidianDistance is really only syntactic sugar as it is side-effect free (it is a "pure function") and can be supported without whole-program analysis by placing several syntactic restrictions on such methods. Equivalently, the expression constituting the return value can be inlined into the guard. Second, however, this expression compared to myRange contains several *attributes* as well as a nested pure function Math .sqrt part of the standard Java class libraries. While EventJava's own substrate supports such attribute expressions as well as the use of standard pure functions, it generates code to interface also with substrates which do not support such expressive subscriptions through corresponding code for local evaluation of the euclidian distance. EventJava currently does not support comparisons of two expressions containing an arbitrary set of constants, attributes, and variables *on both sides of the comparison* because we have not yet encountered applications depending on them.

4.4.3 Switches

Another interesting scenario not captured above is that of a guard predicate based solely on local variables and constants, acting as a "switch" to repeatedly enable/disable a subscription. An example is the first predicate in the guard below:

```
class Broker6 {
  float threshold, balance, tradeLevel;
  event quote(long time, String org, float price)
  when (balance > tradeLevel + 5000 && price >
        threshold) ...
}
```

In fact, the EventJava compiler here will insert code to track updates to all involved variables, but will introduce a virtual boolean variable which simply represents the predicate value. We have extended our substrate to support such boolean variables.

4.4.4 EventJava Substrate

The default substrate for EventJava has three event dissemination modes – a group communication mode and two content-based publish/subscribe (CPS) modes. The group communication mode uses JGroups for the dissemination of events, while the CPS modes uses a CPS system with a broker overlay network, subscription summarization based on subsumption, and the Rete algorithm for matching events to subscriptions in a broker. The difference between the two CPS modes is in the support for expressive dynamic guards – the two modes will be referred to as CPS-Traditional and CPS-Dynamic in the rest of this paper. We will compare the performance of CPS-Traditional and CPS-Dynamic to gauge the benefits due to guard analysis. EventJava with CPS-Traditional uses re-subscriptions for updating guards.

5. CAUSALITY ANALYSIS

Several event processing systems or languages support correlation based on the order of *occurrence* of events. In EventJava, this can be expressed in a guard by writing $e < e'$ where $C.e$ and $C'.e'$ are two event types, declared by C and C' respectively, in the corresponding pattern.

5.1 Causal Dependencies

The order $<$ can be defined by a physical notion of time, or in the absence of synchronized clocks in asynchronous distributed systems, can be based on a logical notion of time such as *causality* [27]. Such a notion in the presence of multicast can be achieved by employing a *causal* order multicast substrate, employing dedicated protocols [37] to avoid jeopardizing safety and/or liveness in the presence of even a single failure. Two factors however complicate the bigger picture: First, these protocols induce a significant overhead compared to un-ordered approaches. Second, other than by the presence of order-based correlation with $<$ in guards, it is impossible in the general case to infer automatically from a program which types of events require causal order (or other ordering guarantees for that matter). The approach of pessimistically conveying every event of a given application through a same causal order multicast group will however lead to a severe bottleneck especially as the number of involved processes increases. At present, programmers thus must explicitly deal with setting up and managing multiple multicast groups, which can involve the same or overlapping sets of application components.

5.2 Local Static Analysis

EventJava thus relies only on programmers to indicate (1) event types whose instances need to be causally *ordered among each other* (which is impossible to infer in general as mentioned), and (2) event types whose respective instances are *causally independent* of each other. By analyzing EventJava programs, the compiler performs a static analysis to infer dependencies among types identified by (1) that are introduced through the program. This ensures safety in the sense of consistency. By reducing the dependencies by (2) a high-overhead pessimistic approach can be avoided. For simplicity we omit in the following subtyping at first. *Direct* dependencies are twofold and trivially inferred:

Direct consume-produce dependency: Any event type $C.e$ produced (e.g., $C.e(\dots)$) directly by a reaction to a pattern involving event type $C'.e'$ implies a consume-produce dependency between instances of $C'.e'$ and of $C.e$.

Direct consume-consume dependency: Any correlated events types $C.e$ and $C'.e'$ appearing in a pattern together (with or without explicit correlation $<$) pessimistically may lead to dependencies among the instances of $C.e$ and of $C'.e'$.

Such consume-produce and consume-consume dependencies can however occur also *indirectly* and *transitively*: for instance, a reaction consuming instances of $C.e$ can through a chain of dependencies involving method invocations (control-flow dependencies) and writes/reads of fields (data-flow) contribute to the generation of an event of type $C'.e'$.

Our static analysis thus computes for every method $C.m$ and reaction to an event $C.e$ (in a pattern), $out(C.m)$ and $out(C.e)$ respectively, which contain the set of events generated, the set of methods invoked, as well as the set of fields written by $C.m$ or $C.e$ respectively. Further the analysis computes $in(C.m)$ and $in(C.e)$, which represent the set of fields read by $C.m$ or $C.e$ respectively.

5.3 Dynamic Closures

Upon loading a class, the runtime computes *locally* the *reflexive transitive closures* $out^+(\dots\dots)$ of the above output sets for that class. The closure of $in(\dots\dots)$ is not needed as the closure of $out(\dots\dots)$ includes all chains of transitive field writes+reads. The same sets are updated for previously loaded classes.

Inheritance is taken into account when computing closures for methods and events. Assume a class C' which extends a class C. For every method m of C overridden by C', $out(C.m) \leftarrow out(C.m) \cup out(C'.m)$.

Now, two events $C.e$ and $C'.e'$ are dependent, noted $C.e \longrightarrow C'.e'$, iff they are directly dependent (see above) or

Indirect consume-produce dependency: there is an indirect consume-produce dependency between $C.e$ and $C'.e'$, i.e., $C'.e' \in out^+(C.e.)$, or

Indirect consume-consume dependency: there is an indirect consume-consume dependency between $C.e$ and $C'.e'$, i.e., there exists a field $C''.f \in (out^+(C.e) \cap out^+(C'.e'))$.

The symmetric binary relation $\longleftrightarrow$ denotes a dependency either way, i.e., $C.e \longleftrightarrow C'.e' \Leftrightarrow C.e \longrightarrow C'.e' \vee C'.e' \longrightarrow C.e$.

A programmer can specify independence $C.e$ `indep` $C'.e'$ to override our pessimistic program analysis. The $\longleftrightarrow_0$ relation is obtained from $\longleftrightarrow$ by removing pairs in `indep` . $\longleftrightarrow^+$ is the reflexive and transitive closure of $\longleftrightarrow_0$. Finally, $\longleftrightarrow^+$ gives rise to *dependency sets* $\overline{d}=d_1,\dots,d_q$ where $\{C.e, C'.e'\} \subseteq d \in \overline{d} \Leftrightarrow C.e \longleftrightarrow^+ C'.e'$

5.4 Runtime Support

Now that we know the local dependencies for every JVM component, we need to coordinate across components. This requires more dedicated runtime support described below.

5.4.1 Group membership

For simplicity we assume in the following a *distributed locking service* like Apache ZooKeeper[1], with a hierarchical name space.

- There is a node `/groups/`*sid* for every dependency set $d = \overline{C.e}$ (i.e., $d = C_1.e_1 \cdot \ldots \cdot C_n.e_n$) where *sid* is the concatenation of the event names $C_i.e_i$ (with a dedicated separation symbol) ordered lexically (we write

[1]http://hadoop.apache.org/zookeeper/

```
1: init
2:   $\overline{d}$                                  {dependency sets in service}
3:   $\overline{d'}$                            {based on local dependency sets}
4:   $\overline{d''} \leftarrow$ sets from reflexive and transitive closure of $\longleftrightarrow_g$
     $\cup \longleftrightarrow'_g$ on $\{C.e \mid C.e \in d_i \vee C.e \in d'_i\}$

5: for all $d''_i$ do
6:   $\overline{d'''} \leftarrow \{d_j \mid d_j \subseteq d''_i\}$
7:   $events \leftarrow \{C.e \mid C.e \in d'''_i\}$
8:   if $|events| > |d'''_1|$ then
9:     $sid \leftarrow sid(d''_i)$
10:    create node /groups/$sid$ with value $gid(sid)$
11:    create group $gid(sid)$
12:    for all $d'''_k$ do
13:      merge $gid(sid(d'''_k))$ to $gid(sid)$
14:      remove node /groups/$sid(d'''_k)$
15:    for all $C.e \in events$ do
16:      if $\exists$ node /events/$C.e$ then
17:        set value of node to $sid(d''_i)$
18:      else
19:        create node with value $sid(d''_i)$
```

Figure 3: Reconciling sets of causally dependent event types across nodes.

$sid = sid(d)$ for brevity). The value gid stored for the node is a group identifier and is given by the hash $hash(sid)$ of the sid of the dependency set. Note that a dependency set may contain only one event.

- There is a node /events/$C.e$ for every event $C.e$. The value stored for the node is the identifer sid of the respective dependency set.

A component has a local hashtable with $\langle C.e, gid\rangle$ entries, where gid is the group identifier for $C.e$.

5.4.2 Group operations

Every time a VM identifies a local change in its dependencies (upon loading of a set of classes), its first locks the node hierarchy. Next it reads all groups from the locking service $\overline{d}$, where each $d_i = \overline{C.e}$. Let $\longleftrightarrow_g$ and $\longleftrightarrow'_g$ represent the symmetric dependency relations (transitive and reflexive closures) underlying $\overline{d}$ and $\overline{d}'$ where the latter represents the local dependency sets.

Let d be the total set of event types across $\overline{d}$ and $\overline{d'}$. Then $\overline{d''}$ is the dependency set constructed from d and the relation $\longleftrightarrow_g \cup \longleftrightarrow'_g$ as described in Figure 3. The *merging* of two multicast groups (see Line 13) happens by having nodes of the two joined groups join the newly formed joint group (at Line 11) individually. To that end, specific merge messages are multicast within the two joined groups.

6. EVALUATION

This section empirically evaluates the benefits of the static analyses presented in the previous section in terms of performance.

6.1 Benefits of Immutability Analysis

To evaluate the benefits of our immutability analysis, we deploy a matcher that correlates events of several types on a single node. We assume that each event has 20 primitive attributes. The matcher correlates 100 event types across 20 correlation patterns, and delivers events that match each of the patterns to a set of sinks. The throughput of the matcher is the number of matched events delivered per second. Table 1 demonstrates the decrease in throughput when matched events are cloned before being delivered to sinks. Table 1 clearly shows the effect, due to cloning, of (1) the number of attributes of events and (2) the number of sinks to which events have to be delivered on the throughput.

# of attributes	# of sinks	% throughput decrease
10	20	3.6%
10	200	13.4%
10	100	42.3%
20	20	6.34%
20	200	15.5%
20	1000	66.7%

Table 1: Benefits of immutability analysis.

6.2 Benefits of Guard Analysis

In this section, we evaluate the benefits of expressive dynamic guards in EventJava.

6.2.1 Metrics

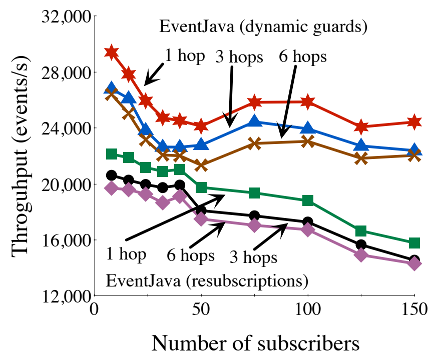
We use four metrics:

(a) **Delay:** Delay is the time period between an update and the reception of the first corresponding event. If a subscriber r_i changes its subscription Φ_i to Φ'_i at time t_0, and the first event method matching Φ'_i but not Φ_i is delivered at time t_1, then the delay at subscriber r_i is defined as t_1-t_0.

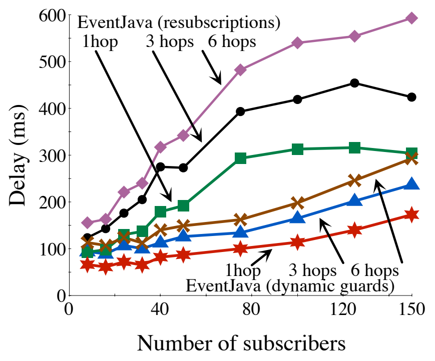
(b) **Throughput:** Throughput is the average number of useful events *delivered* by a subscriber per second. This throughput depends on the number of publishers, event production rates at each publisher, the selectivity of the subscriptions of the subscribers, and the rate at which each subscriber updates its subscriptions. Selectivity of a subscription is the probability that an event matches a subscription. A selectivity of 1.0 implies that a subscription is satisfied by every published event of the respective type and a selectivity of 0.0 implies that none do.

(c) **Spurious events:** The effect of inefficient updates might be offset if brokers are powerful dedicated servers or individual clients are only interested in few events to start with. Increased stress might otherwise manifest, especially on resource-constrained clients. To gauge this stress, we measure the amount of spurious events delivered by clients. If a subscriber r_i changes its subscription Φ_i to Φ'_i at time t_0, then spurious events are those matching Φ_i but not Φ'_i and received by the client *after* t_0 and filtered out locally to it.

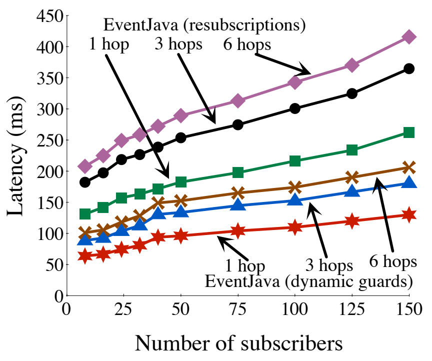
(d) **Latency:** We use the term latency to refer to event dissemination latency: if an event e is produced at time t_1 and is received by a subscriber at time t_2, then the dissemination latency of that event is $t_2 - t_1$. Since we average over a number of runs with the same deployment for all scenarios and systems and the goal is not to measure *exact* latency but rather to gauge (relative) improvements, the clocks of publisher and subscribers



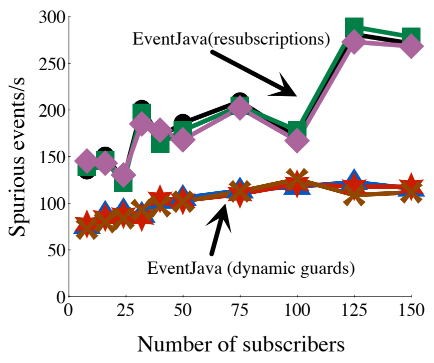
(a) Throughput



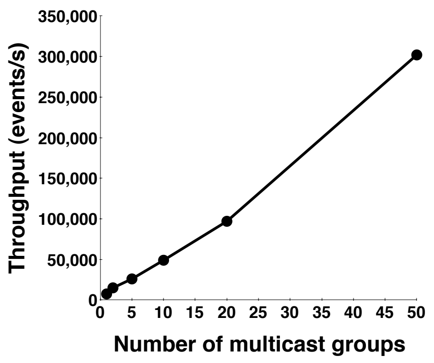
(b) Delay



(c) Latency



(d) Spurious events



(e) Benefits of causality analysis

Figure 4: Benefits of various static analyses. Figures (a) to (d) compare the performance of EventJava with and without support for expressive dynamic guards. Figure (e) shows the benefits of causality analysis.

do not need to be perfectly synchronized. Nonetheless, and regardless of the fact that our routing algorithms described in this paper do not require or directly benefit from synchronized clocks, we ensured that clocks of publishers and subscribers were synchronized for the experiments that measured latency.

6.2.2 Benchmark for guard analysis

We first evaluate the benefits of guard analysis using an artificially generated workload of subscriptions. We use an overlay network of 25 brokers, 30 publishers and up to 150 subscribers. We used an even distribution of $<, \leq, >, \geq, ==$ in the subscriptions, each operator was used in 20% of subscriptions from each subscriber. The broker overlay used for evaluating guard analysis is an undirected acyclic graph. Subscribers are uniformly distributed all over the broker overlay, with maximum number of hops between a publisher and a subscriber being 6. We measure the average throughput, latency, delay and frequency of spurious events at all subscribers with the same number of hops away from any producer. At each subscriber, all the generated subscriptions compared event attributes to expressions on field variables. We did not use any single-field guards because the benefits of using single-field guards has already been exhaustively evaluated by [25]. 50% of the generated subscriptions at each subscriber used arithmetic expressions on variables, while the remaining 50% used functions from `java.lang.Math`.

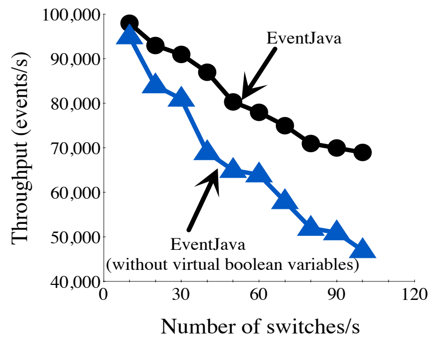
All brokers were executed on dual core Intel Xeon 3.2GHz machines with 4GB RAM running Linux, with each machine executing exactly one broker. Subscribers were deployed on eight core Intel Xeon 1.8GHz machine with 8GB RAM running Linux, with 8 subscribers deployed on each node (one subscriber per core), or on dual code Intel Xeon 3.2GHz machines with 4GB RAM running Linux with 4 subscribers per node. Publishers were deployed on dual core Intel Pentium 3GHz machines with 4GB RAM, with no more than 2 publishers per machine (one publisher per core). Deploying publishers, subscribers and brokers on different nodes ensured that all relevant communication (publisher-broker, broker-broker and subscriber-broker) was over a network, and in many cases across LANs. 10ms delays were added to each network link to simulate wide area network characteristics as is done in EmuLab [2].

6.2.3 Results

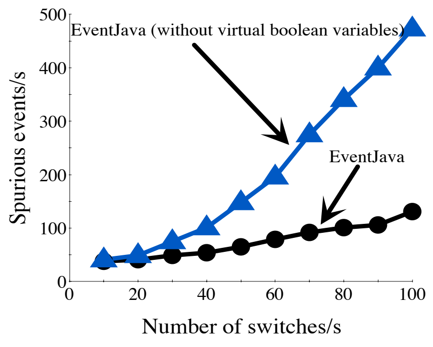
We assess the benefits of dynamic guards, which are facilitated by our guard analysis.

Figure 4(a) shows the difference in throughput between EventJava (with expressive dynamic guards) and EventJava with CPS-Traditional. Figure 4(a) shows that the different in throughput between the two versions of EventJava becomes more pronounced as the number of subscribers increases. Figure 4(a) also shows that the throughput is slightly higher when the number of hops decreases. The average throughput at subscribers located 1 hop away from the publishers is about 1%-4% higher than the average throughput at subscribers located 3 hops away. Similarly, the throughput at subscribers located 3 hops away from the publishers

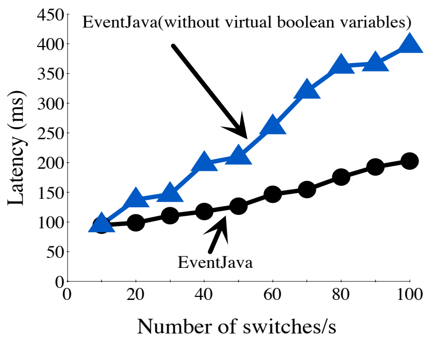
[2]`http://www.emulab.net`



(a) Throughput



(b) Spurious Events



(c) Delay

Figure 5: EventJava with and without boolean guard variables.

is about 9%-12% higher than average throughput at subscribers located 6 hops away.

Figure 4(c) shows the difference in throughput between EventJava (with expressive dynamic guards) and EventJava with CPS-Traditional. Figure 4(c) shows that the difference in latency between the two versions of EventJava increases as the number of subscribers increases. As expected, the event dissemination latency increases as the number of hops increases. The average latency at 3 hops is approximately 28% higher than that at 1 hop, and the average latency at 6 hops is approximately 42% higher that that at 1 hop. A similar trend can be seen in update delay. Figure 4(c) shows that the average delay at 3 hops is approximately 32% higher than the delay at 1 hop, while the delay at 6 hops is approximately 49% higher than the delay at 1 hop.

Figure 4(d) shows that EventJava with resubscriptions receives spurious events at a higher frequency than EventJava with expressive dynamic guards, and this increases the number of subscribers increases. The differences between the average frequencies of spurious events when the distance between the producers and subscribers increases is statistically insignificant – this is because the edge broker immediately applies updates and unsubscriptions to its local data structures. Although the number of spurious events between brokers may increase when the number of hops between the producer and subscriber increases, they are filtered away at the edge broker, and hence the frequency of spurious events received by the subscriber doesn't depend on its distance from the producer.

6.2.4 *Virtual boolean variables*

Figure 5 shows the difference in throughput, number of spurious events and delay between two versions of EventJava – (a) with support in the matcher for boolean variables, and (b) without such support but by (un-)subscribing upon changes in the boolean value.

For this experiment, we used the algorithmic trading benchmark with 1000 event methods and 200 correlation patterns. We also extended our Rete-based matcher to perform distributed matching in a cluster. The matcher divides the correlation patterns uniformly into a n groups if there are n nodes in the cluster, such that each node correlates roughly the same number of event types. Consequently, this experiment doesn't depend on the overlay. We deployed our distributed matcher with and without support for boolean variables on 20 nodes of a cluster of Amazon Elastic Cloud 2 (EC2) large instances (dual core with 7.5 GB memory). Figure 5 shows that the use of boolean variables improves throughput by up to 31.4%, reduces the number of spurious events by up to 3.5× and reduces delay by up to 2×.

6.3 Benefits of Causality Analysis

The objective of causality analysis is to identify as many clusters of events as possible which are not causally related to each other, based on their types. The consequence is that the entire set of application events can be divided into independent sets of events, and using one multicast group per set increases overall throughput. To assess the benefits of conveying unrelated events with independent causal multicast groups we created a varying number of such groups to multicast the same set of events of 500 event types.. To not over-accentuate any trends, we used a lightweight causal ordering protocol based on vector clocks stripped of parts of the actual fault tolerance mechanisms which could have lead to waiting time even in the absence of failures. That way the protocol is not destined upfront to be a bottleneck. Figure 4(e) demonstrates that throughput (of event transmission from source to sinks) increases almost linearly as the number of multicast groups used increases from 1 to 50.

7. RELATED WORK

We summarize related work on support for engineering distributed event-based applications.

7.1 Publish/Subscribe

The benefits of event-based design for distributed software has been long recognized. The path has been led by systems such as JEDI [12] or Siena [10]. Early work on the publish/subscribe architectural pattern [31] targeted enterprise application integration specifically, and thus advocated *self-describing* messages which avoid any agreement on event types or schemas. Much literature on subscription handling at event *brokers* similarly alluded to conformance models by side-stepping the issue of typing events (e.g. [2]) while most respective systems do implement stronger forms of typing. Self-describing messages/structural conformance provide much flexibility yet for many applications designed upfront around the idiom of events nowadays they incur an unnecessary overhead, and bear type safety issues [34]. These are shared by manipulation of SQL-like queries [42].

Several publish/subscribe systems have been extended for correlation (e.g., Gryphon [2], PADRES [28]).

7.2 Programming Languages and Models

Several programming languages support event-based programming. Rather than focusing on the underlying programming paradigms (e.g. objects, functional), we summarize these languages according to three dimensions $\langle k, m, n\rangle$ of expressiveness: 1. the maximum window sizes k for streams of individual event types (k-size windows), 2. the maximum number m of event types that can be correlated ($m-$way joins), and 3. the maximum number of event types n involved, transitively, in a guard ($n-$ary subscriptions). Typically, supporting only intra-event predicates in guards leads to 1-ary subscriptions. Obviously 3. is upper bound by 2. # represents the absence of a bound. We focus on correlation as return values are mostly a syntactic sugar, and the mechanisms for unicast or multicast are better understood currently than syntax and semantics of correlation.

Simple event handlers $\langle 1, 1, 0\rangle$ — provided by most library-based event handlers, the observer design pattern, and simple languages like Ptolemy [35] — support reactions to single instances of multicast events, without guards. nesC [17] is a supports reactions to singleton events, but only for vertical (local, inter-module) interaction (and not horizontal, inter-process, interaction). Languages like CML [36] and Erlang [6] support only *staged* correlation where the occurrence of an event of a first type conditions the consumption of the second one etc. In CML, events are reified as function *evaluations* such as reads or writes on channels, which can be combined. Staged event matching imposes an order on how events are matched to a correlation pattern. This gives the programmer 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.

Simple guarded event handlers $\langle 1, 1, 1\rangle$ include languages inspired by the publish/subscribe paradigm. ECO (*events, constraints, objects*) [18] or Java$_{PS}$ [14] extend C++ and Java respectively. ECO introduces specific first-class constructs to reify events, while Java$_{PS}$ uses objects of specific serializable types. Actor-based languages or libraries supporting guards on *individual* messages such as Erlang [6] are similar yet with unicast.

Chorded languages $\langle 1, \#, 0\rangle$ correspond to the Join Calculus [16] family. Examples are Join Java [23] or Polyphonic C# [7] – now Cω. Chorded languages provide a means to react to correlated asynchronous method invocations, without guards. The only multicast supported is a form of one-*of*-many where several patterns on a *same* object can compete for an event. While Cω is integrated with .NET remoting thus allowing for distributed interaction, most other languages focus on centralized, concurrent systems.

Guarded chorded languages $\langle 1, \#, \#\rangle$ are second generation chorded languages such as JErlang [33] with support for guards with conditions involving n-ary predicates but no streams. Scala Joins [19] represent a special, isolated, case of guarded chorded language supporting only 1-ary predicates on event attributes ($\langle 1, \#, 1\rangle$). Other than that the features provided are as for the common chorded languages.

Generic correlation $\langle \#, \#, \#\rangle$ is to the best of our knowledge only supported by EventJava [15]. Several systems (e.g., Cayuga [13], Borealis [41], StreamBase [1]) implement all correlation features, with unicast. CQL [5] and StreamSQL [1] are SQL derivations with comparable expressiveness. In our earlier work on EventJava[15], we formalize and proves guarantees (e.g., total order) in the presence of correlation. We exploit the language to generate code for efficient correlation of events [24].

Other related languages include the StreamIt [4] dataflow language for fine-grained highly parallel stream applications. 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. StreamFlex [38] is a Java API for stream processing inspired by StreamIt but providing high-predictability implemented on top of a real-time virtual machine. StreamFlex provides *filter*s and (unicast) *channel*s with explicit pull-style consumption of events, leading to a similar staged matching model as CML.

7.3 Aspect-oriented Techniques

Several aspect-oriented programming languages have been proposed for distributed systems programming. As pointed out in [35], joinpoints of AspectJ [26]-like languages can be viewed as expressing *implicit* events as opposed to the explicit events of EventJava and related languages. AWED (*aspects with explicit distribution*) [29]) and DADO (*distributed aspects for distributed objects*) [43] are aspect languages supporting the remote monitoring of distributed applications with distributed pointcuts and advice. DJcutter [30] extends AspectJ with remote joinpoints and pointcuts. At the runtime level, DJcutter has a hardwired centralized aspect-server, which constitutes a bottleneck in a large distributed systems. Implementations of other systems are similarly fixed and lack scalability.

8. CONCLUSIONS

The full potential of the event idiom for programming complex distributed applications has yet to be discovered and exploited. In this paper we have presented a programming language framework guiding developers in writing correct and efficient complex event-based distributed applications without hampering flexibility in the use of existing middleware systems or adaptability of resulting software. We have presented several program (mostly) static program analyses implemented in EventJava, yielding benefits in simplicity, efficiency, and safety. It is straightforward to see that guard analysis eliminates the need for a programmer to *manually* (1)introduce variables to capture expressions on guard fields, (2)examine application code to identify any point at which a guard field's value changes. Causality analysis eliminates the need to manually identify direct and indirect produce-consume/consume-consume relationships between events, and handle dynamically loaded classes. Guard and causality analyses respectively guarantee that no update to a guard field or causal dependency between events is missed, because the compiler generates code to track updates, manage broadcast groups and handle dynamic class loading. We are currently refining the core abstractions of our language framework (e.g., reaction synchronization and threading) and extending their options (e.g., correlation with multiple return values). We are also in the process of designing further analyses.

9. REFERENCES

[1] `www.streambase.com`.

[2] M. Aguilera, R. Strom, D. Sturman, M. Astley, and T. Chandra. Matching Events in a Content-Based Subscription System. In *PODC '99*.

[3] H. Alavi, S. Gilbert, and R. Guerraoui. Extensible Encoding of Type Hierarchies. In *POPL '08*.

[4] S. P. Amarasinghe, M. I. Gordon, M. Karczmarek, J. Lin, D. Maze, R. M. Rabbah, and W. Thies. Language and Compiler Design for Streaming Applications. *International Journal of Parallel Programming*, 33(2-3), 2005.

[5] A. Arasu, S. Babu, and J. Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *The VLDB Journal*, 15(2), 2006.

[6] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall, 2nd edition, 1996.

[7] N. Benton, L. Cardelli, and C. Fournet. Modern Concurrency Abstractions for C#. *ACM TOPLAS*, 26(5), 2004.

[8] G. Bierman, E. Meijer, and W. Schulte. The Essence of Data Access in Cω. In *ECOOP'05*.

[9] J.-P. Briot, R. Guerraoui, and K. Löhr. Concurrency, Distribution and Parallelism in Object-Oriented Programming. *ACM Computing Surveys*, 30(3), 1998.

[10] A. Carzaniga, D. Rosenblum, and A. Wolf. Design and Evaluation of a Wide Area Event Notification Service. *ACM TOCS*, 19(3), 2001.

[11] B. Chin and T. D. Millstein. Responders: Language Support for Interactive Applications. In *ECOOP'06*.

[12] G. Cugola, E. D. Nitto, and A. Fuggetta. The JEDI Event-Based Infrastructure and Its Application to the Development of the OPSS WFMS. *IEEE TSE*, 27(9), 2001.

[13] A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White. Towards Expressive Publish/Subscribe Systems. In *EDBT'06*.

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

[15] P. Eugster and K. Jayaram. EventJava: An Extension of Java for Event Correlation. In *ECOOP'09*.

[16] C. Fournet and C. Gonthier. The Reflexive Chemical Abstract Machine and the Join Calculus. In *POPL'96*.

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

[18] M. Haahr, R. Meier, P. Nixon, V. Cahill, and E. Jul. Filtering and Scalability in the ECO Distributed Event Model. In *PDSE'00*.

[19] P. Haller and T. V. Cutsem. Implementing Joins using Extensible Pattern Matching. In *COORDINATION'09*.

[20] P. Haller and M. Odersky. Scala Actors: Unifying Thread-based and Event-based Programming. *Theoretical Computer Science*, 410(2-3), 2009.

[21] A. Holzer, L. Ziarek, K. R. Jayaram, and P. Eugster. Putting Events in Context: Aspects for Event-based Distributed Programming. In *AOSD'11*.

[22] Y. Huang and H. Garcia-Molina. Parameterized Subscriptions in Publish/Subscribe Systems. *Data Knowl. Eng.*, 60, 2007.

[23] S. V. Itzstein and D. Kearney. The Expression of Common Concurrency Patterns in Join Java. In *PDPTA'04*.

[24] K. R. Jayaram and P. Eugster. Scalable Efficient Composite Event Detection. In *COORDINATION'10*.

[25] K. R. Jayaram, C. Jayalath, and P. Eugster. Parametric Subscriptions for Content-based Publish/Subscribe Networks. In *Middleware'10*.

[26] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An Overview of AspectJ. In *ECOOP'01*.

[27] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *CACM*, 21(7), 1978.

[28] G. Li and H.-A. Jacobsen. Composite Subscriptions in Content-Based Publish/Subscribe Systems. In *Middleware '05*.

[29] L. Navarro, M. Südholt, W. Vanderperren, B. D. Fraine, and D. Suvée. Explicitly Distributed AOP using AWED. In *AOSD'06*.

[30] M. Nishizawa. Remote Pointcut: A Language Construct for Distributed AOP. In *AOSD '04*.

[31] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The Information Bus - An Architecture for Extensible Distributed Systems. In *SOSP'93*.

[32] A. Petrounias and S. Eisenbach. Fairness for Chorded Languages. In *COORDINATION'09*.

[33] H. Plociniczak and S. Eisenbach. Jerlang: Erlang with Joins. In *COORDINATION'10*.

[34] D. Popescu. Impact Analysis for Event-based Components and Systems. In *ICSE'10 Doctoral Symposium*.

[35] H. Rajan and G. T. Leavens. Ptolemy: A Language with Quantified, Typed Events. In *ECOOP'08*.

[36] J. H. Reppy and Y. Xiao. Specialization of CML Message-passing Primitives. In *POPL'07*.

[37] A. Schiper, K. Birman, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM TOCS*, 9, 1991.

[38] J. H. Spring, J. Privat, R. Guerraoui, and J. Vitek. Streamflex: High-throughput Stream Programming in Java. In *OOPSLA'07*.

[39] J. Stokes. *How a stray mouse click choked the NYSE & cost a bank $150K*, 2010.

[40] Sun Microsystems Inc. *Java Message Service - Specification, version 1.1*, 2005.

[41] N. Tatbul, U. Çetintemel, and S. B. Zdonik. Staying FIT: Efficient Load Shedding Techniques for Distributed Stream Processing. In *VLDB'07*.

[42] Z. Tatlock, C. Tucker, D. Shuffelton, R. Jhala, and S. Lerner. Deep Typechecking and Refactoring. In *OOPSLA'08*.

[43] E. Wohlstadter, S. Jackson, and P. Devanbu. DADO: Enhancing Middleware to Support Crosscutting Features in Distributed, Heterogeneous Systems. In *ICSE '03*.