# Decentralized Fault Tolerant Event Correlation

Gregory Aaron Wilkin, Rose-Hulman Institute of Technology
Patrick Eugster, Purdue University
K. R. Jayaram, IBM Thomas J. Watson Research Center, NY

Despite the prognosed use of event correlation techniques for monitoring critical complex infrastructures or dealing with disasters in the physical world, little work exists on making event correlation systems themselves tolerant to failures. Existing systems either provide no guarantees on event deliveries, do not support multicast and thus no guarantees *across* individual processes, or then rely on centralized components or strong assumptions on the infrastructure.

The FAIDECS system attempts to reconcile strong guarantees with practical performance in the presence of process crash failures. To that end, the FAIDECS system uses an overlay network with specific guarantees aligned with its proposed correlation language and guarantees. However, the language proposed lacks expressivity, and the system itself supports only very specific rigid semantics, incapable of supporting even fundamental features like sliding windows.

After providing a comprehensive overview of the FAIDECS model and system, this paper bridges the gap between strong guarantees and more established correlation languages and systems in several steps. First, we propose alternative semantics for several modules of the FAIDECS matching engine and revisit guarantees. Second, we pinpoint which guarantees are contradicted by which combinations of semantic options. Third, we investigate four correlation languages — StreamSQL, EQL, CEL and TESLA — showing which semantic options their respective features correspond to in our model, and thus, ultimately, which guarantees of FAIDECS are maintained by which language features.

Categories and Subject Descriptors: C.2.4 [**Computer Systems Organization**]: Computer Communication Networks—*Distributed Systems*

General Terms: Event, correlation, fault tolerance, agreement, order, guarantee

## 1. INTRODUCTION

Event *correlation* enables higher-level reasoning about interactions in distributed applications by supporting the assembly of *composite* events from elementary ones [Pietzuch et al. 2003; Li and Jacobsen 2005]. Event correlation is widely used in algorithmic trading, intrusion detection [Krügel et al. 2002], network monitoring [Kompella et al. 2005], or many emerging application scenarios. As we detail in Section 7 while presenting related work, most approaches to event correlation exhibit important limitations in *decentralized asynchronous systems prone to crash failures*: (A) no guarantees on composite event deliveries, or (B) no support for multicast and thus no guarantees *across* individual processes; (C) specific architectural setups with centralized components assumed to be reliable or other strong assumptions.

Seminal work on event correlation in the context of *active databases* [Chakravarthy et al. 1994; Gehani et al. 1992; Gatziu and Dittrich 1994], for instance, just like *stream processing* [Balazinska et al. 2008; Demers et al. 2006], considers events to be *uni*cast and focuses on individual processes (cf. B) and centralized correlation engines or components (cf. C). Especially in the presence of failures, processes with the same *subscriptions* may thus receive differing sets and combinations of events (if any at all) and thus reach differing outcomes. Event correlation has also been investigated in the context of *content-based publish/subscribe* systems [Carzaniga et al. 2001] centered on multicast. Examples include Gryphon [Zhao and Strom 2001], PADRES [Li and Jacobsen 2005] and

This research is funded in part by US NSF grants # 0644013 and #0834529, and DARPA grant #N11AP20014.
Author's addresses: Gregory Aaron Wilkin, Computer Science and Software Engineering Department, Rose-Hulman Institute of Technology, Terre Haute, IN 47803; Patrick Eugster, Computer Science Department, Purdue University, West Lafayette, IN 47906; K. R. Jayaram, IBM Thomas J. Watson Research Center, 1101 Kitchawan Rd, Yorktown Heights, NY 10598. Email: jayaramkr@us.ibm.com.

DOI:http://dx.doi.org/10.1145/0000000.0000000

Hermes [Pietzuch et al. 2003]. However, most such extensions focus on efficiency and matching complexity or on the number of possible combinations and thus yield only best-effort guarantees on event delivery (cf. A) unless relying on centralized rendezvous nodes [Pietzuch et al. 2003] (cf. C).

The absence of guarantees or the violation of expectations due to failures can have drastic effects [Sturzrehm et al. 2009]. Consider, for example, monitoring a network to decide which one of two gateways to route certain traffic through. Even if the two gateways receive the same events but in different orders, each gateway might consider itself to be responsible for routing. Worse even, each can consider the other to be responsible. Of course, individual systems can be designed to deal with some of these issues (e.g., by using a proxy process to merge and multiplex streams to replicas), but corresponding solutions are hardly generic and can easily introduce bottlenecks to performance and dependability.

As demonstrated by a wealth of literature, achieving strong guarantees in the presence of failures is a hard problem, even for single event/message delivery scenarios [Défago et al. 2004]. As we showed recently [Wilkin and Eugster 2013], achieving agreement on composite events delivered among processes with identical subscriptions in the presence of process crash failures is as hard as solving the problem of Total Order Broadcast [Hadzilacos and Toueg 1993] on individual events, which in turn is equivalently hard to the fundamental Consensus [Fischer et al. 1985] problem. Intuitively, we considered total order as a suggestion to focus on achieving that in an efficient manner, and proposed FAIDECS (FAIr Decentralized Event Correlation System – "fedex") [Wilkin et al. 2011] which builds specific overlay graphs to consistently merge streams, providing correlation-specific strong guarantees with practical performance. As we demonstrate, this is more efficient than a straightforward solution based on a peer-based (global) total order [Hadzilacos and Toueg 1993] and also more scalable than a less fault tolerant setup with a centralized "sequencer" [Wilkin et al. 2011]. Others have recently proposed solutions to totally order events, albeit layered on top of an order-agnostic overlay [Zhang et al. 2012].

However, the FAIDECS system and model consider very special restricted semantics to achieve strong guarantees. More precisely, only *first received matching* semantics together with *prefix+infix disposal* semantics have been considered thus far: the former means that events from a stream (*type* in FAIDECS) are invariably matched and consumed in their order of reception; the latter means that after matching and delivering an event from a stream, all previously received and still buffered events on the same stream are discarded together with the consumed one. As a consequence, FAIDECS only supports *tumbling* windows on streams of events, but does not support the popular *sliding* windows. In short, FAIDECS thus far provides strong guarantees but with an idiosyncratic model of correlation and subscriptions, making it hard to transpose any results to other systems and languages.

The goal of this paper is to bridge the gap between strong guarantees proposed for FAIDECS and known correlation models and languages. Achieving this goes through several steps: First, we increase the expressiveness of FAIDECS in order to accommodate existing languages. To that end, we present alternative implementations for the *matching* and *disposal* modules of the FAIDECS correlation engine, yielding alternative semantics to the fixed matching and disposal in FAIDECS [Wilkin et al. 2011; Wilkin and Eugster 2013]. Together with some variations of properties, this allows us to express popular features like sliding windows. Second, we investigate properties that are violated by individual combinations of matching and disposal semantics. Third, we map features of existing correlation languages to these semantic options. This allows us to state the properties that are retained by specific operators and features of these languages if the corresponding engine is substituted for that of FAIDECS in the nodes of the FAIDECS overlay network. If we construct complex events by combining operators, intuitively, the set of properties we achieve for the combination is the *intersection* of the properties retained by each of the operators. This paper thus makes the following contributions. After presenting a comprehensive overview of the FAIDECS model [Wilkin and Eugster 2013] and system [Wilkin et al. 2011] (Section 2), we

— increase its expressiveness by describing alternative matching and disposal semantics for its correlation engine (Section 3);

- pinpoint which properties of the FAIDECS model are violated by which combinations of matching and disposal semantics (Section 2.4);
- map four concrete correlation languages — TESLA [Cugola and Margara 2010], StreamSQL [Jain et al. 2008], CEL [Brenna et al. 2007] and EQL[1] — to the semantic framework for FAIDECS, identifying the properties retained by their core operators (Section 5).
- demonstrate the scalability of our decentralized algorithms and explore overall performance benefits and tradeoffs by comparing two different Java implementations of FAIDECS with three different implementations of a global total order of which two are fault tolerant (Section 6).

Section 7 presents related work. Section 8 concludes with final remarks. Our technical report [Wilkin et al. 2014] presents an overview of the well-known StreamSQL, CEL, and EQL languages (while the less popular TESLA language is introduced in Section 5).

## 2. FAIDECS MODEL AND SYSTEM OVERVIEW

This section summarizes the FAIDECS model [Wilkin and Eugster 2013] and system [Wilkin et al. 2011].

### 2.1. System Model and Notation

FAIDECS assumes a system $\Pi$ of *processes*, $\Pi$=$\{p_1$, ..., $p_u\}$, interconnected pairwise by reliable channels [Basu et al. 1996] with primitives to SEND events and receive (RECV) them. The crash-stop failure model is considered [Fischer et al. 1985], i.e., a faulty process may stop prematurely and does not recover. Further, the existence of a discrete global clock is assumed, which processes cannot access. An algorithm run $R$ consists in a sequence of "system" events (not to be confused with the "higher-level" events correlated) on processes. Similar to other models [Aguilera and Toueg 1996], one process thus performs an action per clock tick, which is either of (a) a protocol action (e.g., RECV), (b) an internal action, or (c) a "no-op."

A failure pattern $F$ is a function mapping clock times to processes, where $F(t)$ yields all the processes that crashed by time $t$. Let $crashed(F)$ be the set of all processes $\in$ $\Pi$ that have crashed during $R$. Thus, for a correct process $p_i$, $p_i \in correct(F)$ where $correct(F) = \Pi - crashed(F)$ [Chandra and Toueg 1996].

A formal notation is adopted for properties. Consider the well-known problem of Total Order Broadcast (TOBcast) [Hadzilacos and Toueg 1993] defined over primitives TO-BCAST$(e)$ and TO-DLVR$(e)$ with event $e$. If TO-DLVR$^i(e)_t$ and TO-BCAST$^i(e)_t$ denote the TO-delivery of $e$ by process $p_i$ at time $t$, and the TO-broadcasting of $e$ by $p_i$ at time $t$, respectively, then AGREEMENT [Hadzilacos and Toueg 1993] ("*if some process delivers an event* $e$ *all correct processes eventually deliver* $e$") is defined as follows (note that we elide any of $i, t$, or $e$ when not germane to the context, and write $\exists s$ for a system event $s$ such as a SEND or TO-BCAST as shorthand for $\exists s \in R$): $\exists$TO-DLVR$^i(e) \Rightarrow \forall p_j \in correct(F) \backslash \{p_i\}, \exists$TO-DLVR$^j(e)$

### 2.2. Predicate Grammar

In FAIDECS, ordered sets of delivered events — *relations* — are events aggregated according to specific subscriptions. Such subscriptions are combinations of predicates on events expressed in disjunctive normal form according to the following grammar:

*Subscription* $\Psi ::= \Phi_1 \vee ... \vee \Phi_n$ *Predicate* $\rho ::= T[i].a \; op \, v \mid T[i].a \; op \, T[i].a \mid T[i] \mid \top$
*Conjunction* $\Phi ::= \rho_1 \wedge ... \wedge \rho_m$ *Operation* $op ::= < \mid > \mid \leq \mid \geq \mid = \mid \neq$

A type $T$ can be viewed as a stream of events with identical structure. Such a structure encompasses an ordered set of attributes $[a_1, ..., a_n]$, each of which has a type of its own – typically a scalar type, e.g., `Integer` or `Float`. An event $e$ of type $T$ is an ordered set of values $[v_1, ..., v_n]$ corresponding to the respective attributes of $T$. $T[i].a$ denotes an attribute $a$ of the $i$-th *instance* of type $T$ ($T[i]$)

[1] http://esper.codehaus.org/

– multiple instances of a same type allow *windows* over streams to be captured. $v$ is a value. As syntactic sugar, predicates can refer to just $T.a$, which is automatically translated to $T[1].a$.

A predicate that compares a single event attribute to a value or compares two event attributes on the *same* event, i.e., on the same instance of a same type (e.g., $T_k[i].a \; op \; T_k[i].a'$) is referred to as a *unary* predicate. A *binary* predicate involves two distinct events (two distinct types or different instances of the same type) in a predicate ($T_k[i].a \; op \; T_l[j].a', \; k \neq l \vee i \neq j$). To simplify properties, an *empty* predicate $\top$ is also introduced, which trivially yields *true*. Pointless predicates such as those comparing an attribute to itself ($T_k[i].a \; op \; T_k[i].a$) are prohibited. *Wildcard* predicates of the form $T$ (or $T_k$ for some $k$) simply specify a desired type $T$ of events of interest. $T[i]$ implicitly also declares $T[j] \; \forall j \in [1..i-1]$ if these are not already explicitly declared in the same subscription.

A process $p_j$'s subscription is referred to as $\Psi(p_j)$. By abuse of notation but unambiguously, disjunctions or conjunctions are sometimes handled as sets (of conjunctions and predicates respectively). We write, for instance, $\rho_l \in \Phi \Leftrightarrow \Phi = \rho_1 \wedge ... \wedge \rho_k$ with $l \in [1..k]$, or $\Phi_r \in \Psi \Leftrightarrow \Psi = \Phi_1 \vee ... \vee \Phi_n$ with $r \in [1..n]$. Due to space limitations, and as done in a first step in [Wilkin and Eugster 2013] as well, we focus on subscriptions consisting in *single* conjunctions in the following.

As an example, a subscription for an increase in three successive stock quotes following an earnings report is expressed in the above grammar as:

$\Phi_S = \texttt{SQuote}[0]\texttt{.time} > \texttt{EReport}[0]\texttt{.time} \wedge \texttt{SQuote}[1]\texttt{.value} > \texttt{SQuote}[0]\texttt{.value}$
$\wedge \; \texttt{SQuote}[2]\texttt{.value} > \texttt{SQuote}[1]\texttt{.value}$

### 2.3. Predicate Types and Evaluation

FAIDECS assumes a deterministic order $\prec_N$ within subscriptions based on the names of event types, attributes, etc., which can be used for re-ordering predicates within and across conjunctions. This ordering can be lexical or based on priorities on event types and is necessary for even simplest forms of determinism and agreement. We consider subscriptions to be already ordered accordingly.

The number of events involved in a subscription is given by the number of types and corresponding instances involved. i.e., the types involved in a subscription are represented as *sequences*. As alluded to by the index $i$ in $T[i]$, a same type can be admitted multiple times. Such sequences can be viewed as predicate *signatures*:

$$\begin{array}{lcllcl}
\mathbb{T}(\rho_1 \wedge ... \wedge \rho_m) & = & \mathbb{T}(\rho_1) \uplus ... \uplus \mathbb{T}(\rho_m) & \mathbb{T}(\top) & = & \emptyset \\
\mathbb{T}(T_1[i].a_1 \; op \; T_2[j].a_2) & = & \mathbb{T}(T_1[i]) \uplus \mathbb{T}(T_2[j]) & \mathbb{T}(T[i]) & = & [\underbrace{T, ..., T}_{i\times}] \\
\mathbb{T}(T[i].a \; op \; v) & = & \mathbb{T}(T[i]) & & &
\end{array}$$

$\uplus$ stands for in-order union of sequences defined below ($\oplus$ represents simple concatenation):

$$\emptyset \uplus [T, ...] = [T, ...] \qquad\qquad [T, ...] \uplus \emptyset = [T, ...]$$

$$[\underbrace{T_1, ..., T_1}_{i\times}, T_1', ...] \uplus [\underbrace{T_2, ..., T_2}_{j\times}, T_2', ...] = \begin{cases}
[\underbrace{T_1, ..., T_1}_{i\times}] \oplus ([T_1', ...] \uplus [\underbrace{T_2, ..., T_2}_{j\times}, T_2', ...]) & T_1 \prec_N T_2 \\
[\underbrace{T_2, ..., T_2}_{j\times}] \oplus ([T_2', ...] \uplus [\underbrace{T_1, ..., T_1}_{i\times}, T_1', ...]) & T_2 \prec_N T_1 \\
[\underbrace{T_1, ..., T_1}_{max(i,j)\times}] \oplus ([T_1', ...] \uplus [T_2', ...]) & T_1 = T_2
\end{cases}$$

Any subscription $\Phi$ thus involves a sequence of event types $\mathbb{T}(\Phi)$= $[T_1, ..., T_n]$, where we can have for $i, j \in [1..n], i < j$ such that $\forall k \in [i..j] \; T_k = T_i = T_j$, that is, a subsequence of identical types. These imply each a window of $j - i + 1$ events of the respective type. A subscription is evaluated for an ordered set of events $[e_1, ..., e_n]$, where $e_i$ is of type $T_i$. We assume that types of values in predicates are checked statically with respect to the types of events. $T(e)$ returns the type of a given event $e$. Note that we do not introduce a set of uniquely identified types $\{T_1, T_2, ...\}$. This allows for the set of types to be unbounded, which does not violate the assumptions or properties and keeps notation more brief in that we can use $[T_1, ..., T_k]$ to refer to a sequence of $k$ *arbitrary* types, as opposed to, e.g., $[T_{i_1}, ..., T_{i_k}]$.

The evaluation of a conjunction $\Phi$ on a relation is written as $\Phi[e_1, ..., e_n]$. $e_i.a$ denotes the evaluation of an attribute $a$ on an event $e_i$. Evaluation semantics for predicates are thus defined as:

$$\begin{array}{lcl} (\Phi_1 \vee ... \vee \Phi_n)[e_1, ..., e_n] & = & \Phi_1[e_1, ..., e_n] \vee ... \vee \Phi_n[e_1, ..., e_n] \qquad T[e_1, ..., e_n] = \textit{true} \\ (\rho_1 \wedge ... \wedge \rho_m)[e_1, ..., e_n] & = & \rho_1[e_1, ..., e_n] \wedge ... \wedge \rho_m[e_1, ..., e_n] \qquad \top[e_1, ..., e_n] = \textit{true} \end{array}$$

$$\begin{array}{l}(T[i].a \; op \; v) \\ \quad [e_1, ..., e_n]\end{array} = \begin{cases} e_{k+i-1}.a \; op \; v & T(e_k) = T \wedge (T(e_{k-1}) \neq T \vee (k-1) = 0) \\ \textit{false} & \text{otherwise} \end{cases}$$

$$\begin{array}{l}(T_1[i].a_1 \, op \, T_2[j].a_2) \\ \quad [e_1, ..., e_n]\end{array} = \begin{cases} e_{k+i-1}.a_1 \, op \, e_{l+j-1}.a_2 & T(e_k) = T_1 \wedge (T(e_{k-1}) \neq T_1 \vee (k-1) = 0) \wedge \\ & \quad T(e_l) = T_2 \wedge (T(e_{l-1}) \neq T_2 \vee (l-1) = 0) \\ \textit{false} & \text{otherwise} \end{cases}$$

Parentheses are used for clarity. For brevity, we write simply $\Phi[...]$ for $\Phi[...] = \textit{true}$.

We consider the DLVR primitive to be generically typed, i.e., for delivering a relation $[e_1, ..., e_n]$, we write $\text{DLVR}_\Phi([e_1, ..., e_n])$ where $e_i$ is of type $T_i$ such that $\mathbb{T}(\Phi)$=$[T_1, ..., T_n]$. Analogous to TOBcast, $\text{DLVR}^i_\Phi([..., e, ...])_t$ defines the delivery event of an event $e$ on process $p_i$ in response to $\Phi$ at time $t$ and $\text{MCAST}^i(e)_t$ defines the multicasting of an event $e$ by $p_i$ at time $t$.

## 2.4. Properties

FAIDECS provides primitives MCAST and DLVR, where DLVR is parameterized by a subscription $\Phi$ and delivers relations. From here on, *deliver* refers to DLVR and *multicast* refers to MCAST.

*2.4.1. Basic Safety Properties.* FAIDECS defines three basic safety properties:

NO DUPLICATION $\exists \; \text{DLVR}^i_\Phi([..., e, ...])_t \; \Rightarrow \; \nexists \text{DLVR}^i_\Phi([..., e, ...])_{t'} \mid t' \neq t$

NO CREATION $\exists \text{DLVR}_\Phi([..., e, ...])_t \; \Rightarrow \exists \text{MCAST}(e)_{t'} \mid t' < t$

ADMISSION $\exists \text{DLVR}^i_\Phi([e_1, ..., e_n]) \mid \mathbb{T}(\Phi) = [T_1, ..., T_n] \Rightarrow \; \Phi \in \Psi(p_i) \wedge \Phi[e_1, ..., e_n] \wedge$
$\forall k \in [1..n] : \; T(e_k) = T_k$

NO DUPLICATION implies that a same event is delivered at most once on any single process for a conjunction, which may be opposed to certain systems that allow a same event to be correlated multiple times. We present an alternative property for sliding windows later on.

*2.4.2. Liveness.* ADMISSION can trivially hold while not performing any deliveries. We have to be careful about providing strong delivery properties on *individually* multicast events though, as events may depend on others to match a given conjunction. FAIDECS proposes the two following complementary liveness properties:

CONJUNCTION VALIDITY $\exists \text{MCAST}(e^k_l), k \in [1..n], l \in [1..\infty] \; \wedge \; p_i \in \textit{correct}(F) \; \wedge$
$\exists \Phi \in \Psi(p_i) \mid \Phi[e^1_l, ..., e^n_l] \; \Rightarrow \exists \text{DLVR}^i_\Phi([...])_{t_j} \mid j \in [1..\infty]$

EVENT VALIDITY $\exists \text{MCAST}^i(e^x), \text{MCAST}^{k,l}(e^k_l), k \in [1..n] \backslash x, l \in [1..\infty] \mid$
$\{p_i, p_j, p_{k,l}\} \subseteq \textit{correct}(F) \; \wedge \; \Phi \in \Psi(p_j) \wedge \mathbb{T}(\Phi) = [T_1, ..., T_n] \; \wedge \; \forall z \in [w..y],$
$T_z = T(e^x) \; \wedge \; \nexists (T(e^x)[x - w + 1].a_1 \; op \; T[r].a_2) \in \Phi \mid (T \neq T(e^x) \vee r \neq x - w + 1) \; \wedge$
$\Phi[e^1_l, ..., e^{x-1}_l, e^x, e^{x+1}_l, ..., e^n_l] \Rightarrow \exists \text{DLVR}^j_\Phi([..., e^x, ...])$

These two properties deal with the two possible cases that can arise. The first property deals with dependencies across events and can be paraphrased as follows: "If for a correct process $p_i$ there is an infinite number of relations of matching events that are successfully multicast, then $p_i$ will deliver infinitely many such relations." This property is reminiscent of the FINITE LOSSES property of fair-lossy channels [Basu et al. 1996]. It allows matching algorithms to discard *some* events for practical purposes (e.g., agreement, ordering), yet ensures that when matching events are continuously multicast, a corresponding process will continuously deliver.

EVENT VALIDITY provides a property analogous to validity for single-event/message deliveries (e.g., TOBcast): If an event is multicast by a correct process $p_i$, and its delivery in response to a conjunction on some correct process $p_j$ is not conditioned by binary predicates with other event types, then the event must be delivered by $p_j$ if events of all other types matching each other are continuously multicast. This latter condition is necessary because the delivery of the event even in the absence of binary predicates requires the *existence* of other events.

The condition also ensures that any unary predicates on the respective event type are satisfied. Note that in the case of multiple instances of that type, for each of which there are only unary predicates that match, the property does not force an event to be delivered more than once as the position of the event is not fixed in the implied delivery. The example in Section 2.2 does not contain a unary predicate, and thus is not affected by this property. If the subscription $\Phi_S$ were extended to trigger only if the value of the U.S. dollar is below some value `v` as in $\Phi'_S = \Phi_S \wedge$ `USDollar.value < v`, then any event matching this predicate will be delivered with the entire relation given by $\Phi_S$.

*2.4.3. Agreement.* We now consider a stronger property for relations delivered across processes:
CONJUNCTION AGREEMENT $\exists \text{DLVR}^i_{\Phi}([e_1, ..., e_n]) \Rightarrow \forall p_j \in \mathit{correct}(F) \backslash \{p_i\} \mid \Phi \in \Psi(p_j) :$
$\exists \text{DLVR}^j_{\Phi}([e_1, ..., e_n])$

The uniform CONJUNCTION AGREEMENT property ensures that two correct processes $p_i$ and $p_j$ with identical subscriptions expressed by the conjunction $\Phi$ must deliver the same relation, *without constraining the respective orders of such deliveries*.

FAIDECS also defines a stronger agreement property, which supports *subscription subsumption* on complex events [Wilkin and Eugster 2013], i.e., the recognition of inclusion or covering relationships among subscriptions, a fundamental concept in publish/subscribe systems [Aguilera et al. 1999; Carzaniga et al. 2001; Triantafillou and Economides 2004].

COVERING AGREEMENT $\exists \text{DLVR}^i_{\Phi \wedge \Phi'}([e_1, ..., e_{n,...}]) \mid ((\mathbb{T}(\Phi) = [T_1, ..., T_n]) \cap \mathbb{T}(\Phi')) = \emptyset \Rightarrow$
$\forall p_j \in \mathit{correct}(F) \backslash \{p_i\} \mid \Phi \in \Psi(p_j) : \exists \text{DLVR}^j_{\Phi}([e_1, ..., e_n])$

Formalizing such a property is not trivial because one would also want to retain agreement on (sub-)relations, i.e., that events delivered *together* as part of the more specific subscription are delivered *together* as well for the more generic one. This leads to fundamental limitations. COVERING AGREEMENT only holds for conjunctions which are respectively "extended to the right" with respect to the subscription order $\prec_N$, and the condition on disjointness of the sets of types, e.g., between $\Phi$ and $\Phi'$, makes the sub-conjunctions *independent*.

*2.4.4. Ordering.* FAIDECS defines a number of ordering properties [Wilkin and Eugster 2013], corresponding to the classic FIFO, total, and causal order properties [Hadzilacos and Toueg 1993]. We consider two total order properties:

TYPE TOTAL ORDER $\exists \text{DLVR}^i_{\Phi}([..., e, ...])_{t_i}, \text{DLVR}^i_{\Phi}([..., e', ...])_{t'_i}, \text{DLVR}^j_{\Phi'}([..., e, ...])_{t_j},$
$\text{DLVR}^j_{\Phi'}([..., e', ...])_{t'_j} \mid T(e) = T(e') \Rightarrow (t_i < t'_i \Leftrightarrow \neg(t'_j < t_j))$

CONJUNCTION TOTAL ORDER $\exists \text{DLVR}^i_{\Phi \wedge \Phi'}([e_1, ..., e_n, ...])_{t_i}, \text{DLVR}^i_{\Phi \wedge \Phi'}([e'_1, ..., e'_n, ...])_{t'_i},$
$\text{DLVR}^j_{\Phi \wedge \Phi''}([e_1, ..., e_n, ...])_{t_j}, \text{DLVR}^j_{\Phi \wedge \Phi''}([e'_1, ..., e'_n, ...])_{t'_j} \mid$
$((\mathbb{T}(\Phi) = [T_1, ..., T_n]) \cap \mathbb{T}(\Phi')) = \emptyset \wedge (\mathbb{T}(\Phi) \cap \mathbb{T}(\Phi'')) = \emptyset \Rightarrow (t_i < t'_i \Leftrightarrow t_j < t'_j)$

TYPE TOTAL ORDER ensures that there is a total (sub-)order on the messages of a same type. CONJUNCTION TOTAL ORDER ensures that (sub-)*relations* delivered to identical (sub-)conjunctions are delivered in a total order. An implementation which *never* enforces CONJUNCTION TOTAL ORDER, i.e., delivers no two same relations on two processes with identical (sub-)conjunctions, could still ensure TYPE TOTAL ORDER. Inversely, TYPE TOTAL ORDER does not imply CONJUNCTION TOTAL ORDER.

Executed by every $p_i$.

```
init
    $\Psi \leftarrow \Phi$
    $\Phi \leftarrow \rho_1 \wedge \ldots \wedge \rho_m$
    $Q[T] \leftarrow \emptyset$
    SEND(SUB, $\Phi$) to PROCESS($\sqcup\mathbb{T}(\Phi)$)
To MCAST($e$):
    SEND(EVENT, $e$) to PROCESS($[T(e)]$)
function MATCH ($[e'_1, ..., e'_n], \Phi, Q$)
    $T \leftarrow T_{n+1} \mid \mathbb{T}(\Phi) = [T_1, ..., T_{n+1}, ...]$
    $l \leftarrow max(j \mid Q[T] = e_1 \oplus ... \oplus e_j \oplus ... \oplus e_h) \mid$
      $\exists k \in [1..n] : e_j = e'_k$
    for all $k = (l+1)..h$ do
        if $|\mathbb{T}(\Phi)| = n+1$ then
            if $\Phi[e'_1, ..., e'_n, e_k]$ then
                return $[e'_1, ..., e'_n, e_k]$
        else
            $E \leftarrow$ MATCH($[e'_1, ..., e'_n, e_k], \Phi, Q$)
            if $E \neq \emptyset$ then
                return $E$
    return $\emptyset$
upon RECV(EVENT, $e$) do
    if ENQUEUE($e, \Phi, Q$) then
        $[e_1, ..., e_l] \leftarrow$ MATCH($\emptyset, \Phi, Q$)
        if $l > 0$ then
            DEQUEUE($[e_1, ..., e_l], Q$)
            DLVR$_\Phi$ ($[e_1, ..., e_l]$)
function ENQUEUE ($e, \Phi, Q$)
    $win \leftarrow max(j \mid \exists ... T(e)[j].a ... \in \Phi)$
    if $\forall j = 1..win\ ((\exists \rho = (T(e)[j].a\ op\ v) \in$
      $\Phi \mid \neg\rho[e]) \vee (\exists(\rho = T(e)[j].a\ op$
      $T(e)[j].a') \in \Phi \mid \neg\rho[e]))$ then
        return false
    else
        $Q[T(e)] \leftarrow Q[T(e)] \oplus e$
        return true
procedure DEQUEUE($[e_1, ..., e_m], Q$)
    for all $Q[T] = ... \oplus e_k \oplus e \oplus ..., k \in [1..m]$ do
        $Q[T] \leftarrow e \oplus ...$
```

Fig. 1: First received (FR) matching with prefix+infix (PI) disposal.

Similarly to TYPE TOTAL ORDER, the following property depends on the equivalence of event types among ordered events:

<u>TYPE FIFO ORDER</u> $\exists \text{MCAST}^i(e)_{t_i}, \text{MCAST}^i(e')_{t'_i}, \text{DLVR}^j_\Phi([..., e, ...])_{t_j}, \text{DLVR}^j_\Phi([..., e', ...])_{t'_j} \mid$
$T(e) = T(e') \ \wedge \ t_i < t'_i \ \Rightarrow \ t_j \leq t'_j$

The following property yields a type-specific form of causal order for relations when combined with TYPE FIFO ORDER (like LOCAL ORDER and FIFO ORDER for single-event deliveries [Hadzilacos and Toueg 1993]):

<u>TYPE LOCAL ORDER</u> $\exists \text{DLVR}^i_\Phi([..., e, ...])_{t_i}, \text{MCAST}^i(e')_{t'_i}, \text{DLVR}^j_{\Phi'}([..., e, ...])_{t_j},$
$\text{DLVR}^j_{\Phi'}([..., e', ...])_{t'_j} \mid T(e) = T(e') \ \wedge \ t_i < t'_i \ \Rightarrow \ t_j \leq t'_j$

### 2.5. Decentralized System

FAIDECS implements the above properties with much better scalability than centralized sequencers or peer-based Consensus approaches [Hadzilacos and Toueg 1993], and inherently better fault-tolerance than a sequencer-based approach. The solution assumes a distributed hashtable (DHT) for uniquely identifying processes for given "roles." Lightweight replication mechanisms of such roles are used for reliabiliy.

*2.5.1. Mergers.* All processes with conjunctions on a sequence of event types $[T_1, ..., T_k]$ send their subscriptions to a same process, identified as $p_j$=PROCESS($\sqcup[T_1, ..., T_k]$), responsible for handling all conjunctions on the involved sequence of types *without duplicates*[2]:

$$\sqcup[T_1, ..., T_1, T_2, ...] \ = \ [T_1] \oplus \sqcup[T_2, ...]$$

The function PROCESS relies on a DHT to deterministically identify such responsible processes, called *mergers*. Lodged at the root of the thereby created overlay network (see Figure 2) are mergers responsible for individual event types $T_1$, $T_2$, etc. To ensure the properties with respect to extensions of conjunctions to the right, events undergo an *ordered merge by type* where a merger $p_j$=PROCESS($\sqcup[T_1, ..., T_k]$) gets events of types $T_1$, ..., $T_k$ from two processes: those identified as PROCESS($\sqcup[T_1, ..., T_{k-1}]$) and PROCESS($[T_k]$). Mergers are replicated in FAIDECS to increase

[2] Different processes could be used but deduplication simplifies the algorithm [Wilkin et al. 2011].

fault tolerance, which emphasizes the focus on total order as opposed to FIFO order (which would trivially solve the former in the absence of multiple destinations).

*2.5.2. Clients.* The core constitutents of the algorithm in Figure 1 which performs full correlation at subscribers based on merged streams are two-fold: (1) *matching* (MATCH, Line 8) and (2) *disposal* (DEQUEUE, Line 33). The presented implementations of these modules provide *first received* (FR) matching and *prefix+infix* (PI) disposal respectively [Wilkin et al. 2011; Wilkin and Eugster 2013]. In short, the former means that events are matched on a process in the order received by that process. The latter implies the following: Upon a successful match $[e_1, \ldots, e_n]$, for each event $e_i$, all events of the same type received prior to $e_i$ are discarded via the garbage collection mechanism DEQUEUE. Each process $p_i$ maintains one queue $Q$ per event type in its conjunction $\Phi$=$\Psi(p_i)$. For example, for a conjunction $\Phi = \rho_1 \wedge \rho_2$ where $\rho_1 = T_1.a_1 < T_2.a_2$ and $\rho_2 = T_1.a_1 < 20$, the subscriber maintains one queue for events of type $T_1$ and one for events of type $T_2$. When receiving an event, $p_i$ will check if the type of the event is in $p_i$'s subscription. If so, $p_i$ attempts to ENQUEUE the event. $Q[T(e)] \oplus e$ denotes the appending of event $e$ to the queue of type $T(e)$. The ENQUEUE primitive returns $true$ if the event has been ENQUEUEd, meaning it satisfies all unary predicates on the respective types in the conjunction. Then $p_i$ proceeds to MATCHing. Any single received event may complete up to one relation. If a match $[e_1, \ldots, e_n]$ is identified, the corresponding events are discarded (DEQUEUE) and for each event $e_i$, *all* preceding events of the same type are discarded from the respective queue for that type. MATCH iterates through the queues deterministically. The semantics attempt to find the *first* instance of the first type in $\Phi$ for which there are events of the remaining types with which $\Phi$ is satisfied. Among all such possibilities, the algorithm recursively seeks for a match with the *first* instance of the second type in $\Phi$, etc., until a match is found or all possibilities are exhausted. For multiple instances of a same type, a first instance is recursively matched with the *first follow-up instance* in the same queue until the needed number of instances is found for that type or the queue is exhausted.

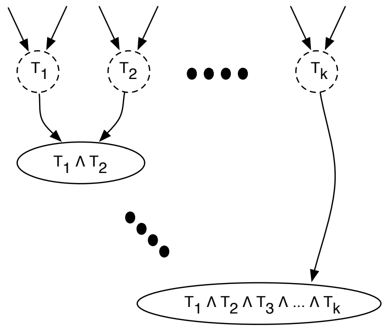


Fig. 2: Overlay for conjunctions. Streams merging follows $\prec_N$

[Wilkin and Eugster 2013] shows how the algorithm of Figure 1 ensures all properties previously outlined. Obviously, there are more efficient ways to implement matching and disposal semantics and will be presented later in Section 5 when we present other correlation languages and also in Section 6 when evaluating the FAIDECS overlay network using these correlation languages.

## 3. SEMANTIC OPTIONS

This section presents semantic alternatives to the default FAIDECS matching algorithm of Figure 1. For the purpose of this section, we will use an example to demonstrate the different semantics described below. For this example, suppose a process $p_1$ has a queue for type $T_1$ such that $Q[T_1]$ = $\{e_1, e_2, e_3, e_4, e_5\}$ and a second queue for type $T_2$ such that $Q[T_2]$ = $\{e_a, e_b, e_c, e_d\}$ at some instant in time.

### 3.1. Event Matching Semantics

The algorithm of Figure 1 makes use of first received *non-contiguous* (FR) matching. In this case, events in each respective queue are considered in the FIFO order for matching. (In the example queues above, $p_1$ would thus consider $e_1$ for a match before $e_2$, and so on within the queue $Q[T_1]$ with this type of matching and $e_a$ before $e_b$ for queue $Q[T_2]$. Note, with non-contiguous matching, $e_1$ and $e_3$ could appear in the same relation without $e_2$.) However, in real-time systems and algorithmic stock trading, which require the most up-to-date information, first received matching may not be the most efficient matching when more recent events tend to be the most pertinent. In this instance, *most-recently received* (MR) matching may be a preferred matching semantic: when an event is received, the *last* instance of an event of a first type is matched with the last found instance of the next, etc., moving backwards in the queues as necessary until either a match is found, or all queues are exhausted. (In the example queues above, $p_1$ would thus scan $Q[T_1]$ starting with $e_5$, then $e_4$ for type $T_1$ and correspondingly $e_d$ first for the queue $Q[T_2]$.) Figure 3 provides the MATCH function for most-recently received *non-contiguous* (MR) matching, which replaces MATCH of Figure 1. As mentioned, Figure 1 is an exhaustive search, thus the following extensions are presented for readability rather than efficiency. Later, in Section 5, we present three correlation languages with more efficient matching semantics.

The matching is thus still *non-contiguous*, meaning that if more than one event of a same type is matched, these events are not guaranteed to be consecutive events from the queue, but rather may be interleaved by other events in the queue. Some applications may also require that matched events of the same type (i.e., from the same stream) are matched in a *contiguous* manner (meaning, for instance, that if $e_1$ and $e_3$ were to appear in the same relation, either $e_2$ *must* also appear in that relation, or it is not considered a match). Figure 4 shows first received *contiguous* (FRC) matching while Figure 5 shows most-recently received *contiguous* (MRC) matching. Both MATCH functions assure that a first found instance of an event is only matched with the next consecutive event if possible.

### 3.2. Event Consumption Semantics

The needs of applications may dictate also how events are discarded/consumed when relations are delivered. There are four main possibilities (suppose, for the following, from the queues $Q[T_1]$ and $Q[T_2]$ given in Section 3, that a relation $[e_2, e_4, e_b, e_c]$ is delivered by process $p_1$ for the following semantics).

Prefix+infix (PI) disposal is the default disposal semantics shown in Figure 1. It discards all events which have been consumed and all events which have been received prior to the last matched event in each respective type queue. Many events which have never been delivered may be discarded. With this type of disposal semantics, if the above relation is delivered, then $Q[T_1]$ will then contain $\{e_5\}$ and $Q[T_2]$ will contain $\{e_d\}$.

*Infix only* (I) disposal exclusively discards events which have been consumed, i.e., delivered as part of a relation. Undelivered events remain in the queue until they are delivered. This is shown by the DEQUEUE function of Figure 6 which replaces that of Figure 1. With this type of disposal semantics, when the above relation is delivered, then $Q[T_1]$ will contain $\{e_1, e_3, e_5\}$ and $Q[T_2]$ will contain $\{e_a, e_d\}$.

*Infix+postfix* (IP) disposal discards all events which have been consumed and all currently queued events received *after* these delivered events. Again, many events which have never been delivered may be discarded. This disposal semantic may allow for an alert of some occurrence of interest, but can eliminate repetitive alerts when only one is desired in a certain time frame. IP disposal is demonstrated by the DEQUEUE function of Figure 7, replacing that of Figure 1. In this case, if the above relation is delivered, then $Q[T_1]$ will contain $\{e_1\}$ and $Q[T_2]$ will contain $\{e_a\}$.

Lastly, in what we call *first prefix* (FP) disposal, every event in each type queue which appears *before* the *first* matched event is discarded along with the very first matched event. As will be shown, this type of disposal is tailored to sliding windows. FP disposal is shown in Figure 8. Here, if the above relation is delivered, then $Q[T_1]$ will contain $\{e_3, e_4, e_5\}$ and $Q[T_2]$ will contain $\{e_c, e_d\}$.

### 3.3. Windows

Much like in stream processing, along with reasoning about the above in terms of matching and disposal semantics, events can be grouped together and discarded according to the current "window" in which events may be matched. If $T_k[i]$ is the largest $i$ for type $T_k$ occurring in a predicate, then the subscription involves a window of size $i$. A window may be viewed as moving forward as time progresses, as events are received or as events are delivered, allowing a certain number, or subset, of

Replaces Lines 8-19 of Figure 1.

```
function MATCH ($[e'_1, ..., e'_n], \Phi, Q$)
    $T \leftarrow T_{n+1} \mid \mathbb{T}(\Phi) = [T_1, ..., T_{n+1}, ...]$
    $l \leftarrow min(j \mid Q[T] = e_1 \oplus ... \oplus e_j \oplus ... \oplus e_h) \mid$
        $\exists k \in [n..1] : e_j = e'_k$
    for all $k = (l-1)..1$ do
        if $|\mathbb{T}(\Phi)| = n + 1$ then
            if $\Phi[e'_1, ..., e'_n, e_k]$ then
                return $[e'_1, ..., e'_n, e_k]$
        else
            $E \leftarrow$ MATCH$([e'_1, ..., e'_n, e_k], \Phi, Q)$
            if $E \neq \emptyset$ then
                return $E$
    return $\emptyset$
```

Fig. 3: MR matching.

Replaces Lines 8-19 of Figure 1.

```
function MATCH ($[e'_1, ..., e'_n], \Phi, Q$)
    $T \leftarrow T_{n+1} \mid \mathbb{T}(\Phi) = [T_1, ..., T_{n+1}, ...]$
    if $T_n \neq T_{n+1}$ then                    {if this type is a new type}
        $h \leftarrow |Q[T]|$
        $l \leftarrow 1$
    else                  {look only to the contiguously next event}
        $l \leftarrow max(j \mid Q[T] = e_1 \oplus ... \oplus e_j \oplus ... \oplus e_h) \mid$
            $\exists k \in [1..n] : e_j = e'_k$
        $h \leftarrow l + 1$
        $l \leftarrow l + 1$            {assure loop only looks at next event}
    for all $k = l..h$ do
        if $|\mathbb{T}(\Phi)| = n + 1$ then
            if $\Phi[e'_1, ..., e'_n, e_k]$ then
                return $[e'_1, ..., e'_n, e_k]$
        else
            $E \leftarrow$ MATCH$([e'_1, ..., e'_n, e_k], \Phi, Q)$
            if $E \neq \emptyset$ then
                return $E$
    return $\emptyset$
```

Fig. 4: FRC matching.

Replaces Lines 8-19 of Figure 1.

```
function MATCH ($[e'_1, ..., e'_n], \Phi, Q$)
    $T \leftarrow T_{n+1} \mid \mathbb{T}(\Phi) = [T_1, ..., T_{n+1}, ...]$
    if $T_n \neq T_{n+1}$ then                    {if this type is a new type}
        $h \leftarrow |Q[T]|$
        $l \leftarrow 1$
    else                  {look only to the contiguously next event}
        $l \leftarrow min(j \mid Q[T] = e_1 \oplus ... \oplus e_j \oplus ... \oplus e_h) \mid$
            $\exists k \in [n..1] : e_j = e'_k$
        $h \leftarrow l - 1$
        $l \leftarrow l - 1$            {assure loop only looks at next event}
    for all $k = h..l$ do                            {loop backwards}
        if $|\mathbb{T}(\Phi)| = n + 1$ then
            if $\Phi[e'_1, ..., e'_n, e_k]$ then
                return $[e'_1, ..., e'_n, e_k]$
        else
            $E \leftarrow$ MATCH$([e'_1, ..., e'_n, e_k], \Phi, Q)$
            if $E \neq \emptyset$ then
                return $E$
    return $\emptyset$
```

Fig. 5: MRC matching.

Replaces Lines 33-35 of Figure 1.

```
procedure DEQUEUE($[e_1, ..., e_m], Q$)
    for all $Q[T] = ... \oplus e_i \oplus e_k \oplus e \oplus ..., k \in [1..m]$ do
        $Q[T] \leftarrow ... \oplus e_i \oplus e \oplus ...$
```

Fig. 6: I disposal.

Replaces Lines 33-35 of Figure 1.

```
procedure DEQUEUE($[e_1, ..., e_m], Q$)
    for all $Q[T] = ... \oplus e \oplus e_k \oplus ..., k \in [1..m]$ do
        $Q[T] \leftarrow ... \oplus e$
```

Fig. 7: IP disposal.

Replaces Lines 33-35 of Figure 1.

```
procedure DEQUEUE($[e_1, ..., e_m], Q$)
    $Q[T] \leftarrow e_2 \oplus ...$
```

Fig. 8: FP disposal (sliding window).

events to be considered for matching at any one instance. When the window has passed events, these events may be discarded, while only events within the window may be considered for matching.

*Tumbling windows* consider a number of events, and when the window is to move forward, it "tumbles" to the next set of events in the queue, which is a completely new set, i.e., no events are considered more than once. In FAIDECS, the disposal semantics (i.e., PI disposal) equate to that of a tumbling window: The window starts as a single event per type, and events are added to the window when a match is not found. After a match is found, the window tumbles over to the immediate next set of events in the respective queues, which have not yet been considered.

*Sliding windows* are common in stream processing. Most commonly, a sliding window considers a fixed number of events, and moves forward by one event at a time as it progresses. Within a window, events may be matched so that they are contiguous, i.e., if more than one event is used from the same window for a single operation, each event must have been immediately received after the previous in the set. In other variations, events may be matched that are non-contiguous, as long as they are each a part of the same window. Sliding windows allow for a same message to be matched more than once in multiple relations, which immediately violates the NO DUPLICATION property given above. Another variation of the property could allow for a single event to be delivered more than once, but never in the same position within two different relations, for instance. A variation of the property which allows for sliding windows is as follows.

No Duplication′ $\exists\, \text{DLVR}^i_\Phi([e_1, ..., e_n])_t \;\Rightarrow \nexists \text{DLVR}^i_\Phi([e'_1, ..., e'_n])_{t'} \mid e_j = e'_j \wedge t' \neq t$

In the case for correlation, sliding windows might be implemented slightly differently. Firstly, as in the FAIDECS algorithm, there could be one window per type. And instead of moving a window one event per round when an event is received, a window might start at the beginning of a match for each type, and then once the corresponding relation is delivered, move each window per type queue by one event. This would assure that no event is delivered twice in the same position of a window, thus ensuring No Duplication′. The above described sliding window is equivalent to FP disposal found in Figure 8.

## 4. PROPERTIES OF SEMANTIC OPTIONS

In this and the following sections, we discuss the unmet properties by comparing matching semantics and disposal semantics. Table I enumerates the properties violated for various combinations of matching and disposal semantics.

| | FR matching | FRC matching | MR matching | MRC matching |
|---|---|---|---|---|
| I disposal | Type Total Order (6)<br>Type FIFO Order (6)<br>Type Causal Order (6) | Event Validity (5)<br>Conjunction Validity (5)<br>Type Total Order (6)<br>Type FIFO Order (6)<br>Type Causal Order (6) | Event Validity (1)<br>Covering Agreement (2)<br>Type Total Order (6)<br>Type FIFO Order (3, 6)<br>Type Causal Order (4, 6) | Event Validity (1, 5)<br>Conjunction Validity (5)<br>Covering Agreement (2)<br>Type Total Order (6)<br>Type FIFO Order (3, 6)<br>Type Causal Order (4, 6) |
| PI disposal (Tumbl. w.) | (All properties met<br>as shown previously [Wilkin et al. 2011]) | Event Validity (5)<br>Conjunction Validity (5) | Event Validity (1)<br>Covering Agreement (2) | Event Validity (1)<br>Conjunction Validity (5)<br>Covering Agreement (2) |
| IP disposal | Event Validity (7)<br>Covering Agreement (8) | Event Validity (5, 7)<br>Conjunction Validity (5)<br>Covering Agreement (8) | Event Validity (1)<br>Covering Agreement (2)<br>Type FIFO Order (3)<br>Type Causal Order (4) | Event Validity (1)<br>Conjunction Validity (5)<br>Covering Agreement (2)<br>Type FIFO Order (3)<br>Type Causal Order (4) |
| FP disposal (Sliding w.) | No Duplication (9)<br>Type FIFO Order (10)<br>Type Causal Order (10) | No Duplication (9)<br>Event Validity (5)<br>Conjunction Validity (5) | No Duplication (9)<br>Event Validity (1)<br>Covering Agreement (2)<br>Type FIFO Order (3, 10)<br>Type Causal Order (4, 10) | No Duplication (9)<br>Event Validity (1)<br>Conjunction Validity (5)<br>Covering Agreement (2)<br>Type FIFO Order (3)<br>Type Causal Order (4) |

Table I: Table of semantic options specifying which properties are not met with applicable theorems in parentheses. Shaded area indicates default semantics for FAIDECS.

### 4.1. First Received vs. Most-Recently Received

Since FAIDECS uses non-contiguous FR matching (with PI disposal) and all of the above properties are met (as shown in the shaded box of Table I), it is clear that taken by itself, FR matching does not violate any properties. Only when non-contiguous FR matching is paired with different disposal semantics are any properties violated. On the contrary, with MR matching, there is *no* combination with disposal semantics that *does not* violate some properties. Particularly, MR matching always violates Event Validity and Covering Agreement. Further, aside from using PI disposal, MR matching violates Type FIFO Order and Type Causal Order.

*4.1.1. Most-Recently Received Matching.* The following Theorems 1–4 prove that MR matching violates several properties.

**Theorem** 1. *MR matching violates* Event Validity

PROOF. This proof will be by counter-example. Suppose a process $p_i$ has a subscription over three event types $T_1$, $T_2$ and $T_3$ such that $\Phi = T_1.a_1 = v \wedge T_2.a_1 < T_3.a_1$. Now suppose that an event $e^1_1$ such that $e^1_1.a_1 = v$ is received, thus qualifying as an event to which EVENT VALIDITY applies. However, due to the lack of other matching events of type $T_2$ and $T_3$, this event $e^1_1$ is not delivered as part of a relation. As more events are received, it is possible that more events of type $T_1$ that match the respective unary predicate are received than may be delivered before matching events of types $T_2$ and $T_3$ are received. As matching events of types $T_2$ and $T_3$ are received, they are then matched with the newer events of type $T_1$. In this case, $e^1_1$ is essentially "buried" and is never again viewed for another possible match since the newer events only are considered, thus EVENT VALIDITY may be violated. □

As an example, consider the following subscription for some arbitrary value of the US dollar of 1: $\Phi_S = \texttt{SQuote}[0]\texttt{.time} > \texttt{EReport}[0]\texttt{.time} \wedge \texttt{SQuote}[1]\texttt{.value} > \texttt{SQuote}[0]\texttt{.value}$ $\wedge\ \texttt{SQuote}[2]\texttt{.value} > \texttt{SQuote}[1]\texttt{.value} \wedge \texttt{USDollar.value < 1}$
In this case, it is possible that an event of type `USDollar` could be received with value 0.74 and then placed in the buffer to await a match with three successive stock quotes of increasing value. However, there may be many events received of type `SQuote` (which are not successively increasing) along with many other events of type `USDollar` before the first three conditions are met. Once three successively increasing events of type `SQuote` are received, there may be a large number of events in the buffer of type `USDollar` that are less than 1 which qualify first to be matched with the stock quote events since the most recent events are desired here. If more events are being received than there are relations being delivered, since MR is used, the first received event of type `USDollar` with value 0.74 may never be used in a match, and thus EVENT VALIDITY is not met.

**THEOREM** 2. *MR matching violates* COVERING AGREEMENT

PROOF. The following proof is by counter-example. Suppose a process $p_i$ has a conjunction $\Phi_i = T_1.a_1 < T_2.a_1$ and another process $p_j$ has a conjunction $\Phi_j = \Phi_i \wedge T_3.a_1 < z$. In this example, now suppose that both $p_i$ and $p_j$ receive two events $e^1_1$ and $e^2_1$ such that $e^1_1.a_1 = v$ and $e^2_1.a_1 = v'$ (s.t. $v < v'$). In this case, both $e^1_1$ and $e^2_1$ match $\Phi_i$, thus process $p_i$ may deliver the relation $[e^1_1, e^2_1]$. However, process $p_j$ must wait for a matching event of type $T_3$ before it may deliver any relations. Now suppose that both $p_i$ and $p_j$ receive a third message $e^2_2$ such that $e^2_2.a_1 = u$ (s.t. $u > v'$). Now, process $p_j$ could receive an event $e^3_1$ such that $e^3_1.a_1 = w$ (s.t. $w < z$). When process $p_j$ triggers a match, it will view the most recent events by the most-recently received matching function, and thus the relation $[e^1_1, e^2_2, e^3_1]$ is delivered which violates COVERING AGREEMENT since process $p_i$ matched $e^1_1$ with $e^2_1$ but process $p_j$ matched $e^1_1$ with $e^2_2$. □

When PI disposal is not used, MR matching may also violate a number of ordering properties, namely TYPE FIFO ORDER and TYPE CAUSAL ORDER.

As an example, consider the subscription $\Phi_S$ above except that one process is only looking for three successive stock quotes, and the second process has the same constraints but also has the last constrain above where `USDollar.value < 1`. In this case, it is possible that three successive stock quotes are published and the first process would deliver them. However, if the second process has not received any events of type `USDollar` with value less than 1, no relations may be delivered by the second process. If a fourth successive stock quote is received, followed by an event of type `USDollar` with value 0.89, then the *last* three stock quote events received (by MR) will be matched with this new `USDollar` event. Thus, the two processes will not agree on the three stock quote events that are delivered since the first process delivered the *first* three stock quote events received, while the second delivered the last three of four received.

**THEOREM** 3. *MR matching with the absence of PI disposal violates* TYPE FIFO ORDER

PROOF. Since events are matched backwards in the queue, it is clear that if some later message $e^k_j$ is matched and delivered in a relation before an earlier event $e^k_i$ such that $i < j$, and some event other than the type $T(e_j^k)$, say $e_l^m$, is then later received, the earlier event $e_i^k$ such that $i < j$ in the same queue as $e_j^k$ might be matched with $e_l^m$ thus violating TYPE FIFO ORDER since $e_j^k$ is delivered *before* $e_i^k$ and $i < j$. □

Consider a more simple subscription: $\Phi_T = \texttt{SQuote}[0].\texttt{time} > \texttt{EReport}[0].\texttt{time} \wedge \texttt{SQuote}[0] \wedge \texttt{USDollar.value < 1}$ which is looking for any stock quote after an earnings report when the US dollar drops below the value 1. In this case, if two events of type `USDollar` are received, both with values less than 1, before any stock quotes are received after an earnings report, it will be the case by MR that the *second* `USDollar` event will eventually be delivered first. Without PI disposal, the first `USDollar` event remains in the queue and can later be delivered, violating TYPE FIFO ORDER.

**THEOREM** 4. *MR matching with the absence of PI disposal violates* TYPE CAUSAL ORDER

PROOF. Without FIFO order, there cannot be causal order in this instance. Thus, it follows by Theorem 3 that TYPE CAUSAL ORDER is violated. □

PI disposal rectifies the issues in Theorems 3–4 since if $e_j^k$ were delivered, the event $e_i^k$ such that $i < j$ would be thus discarded and never delivered and FIFO order would still hold.

The reason why TYPE TOTAL ORDER is violated for MR matching with I disposal will be explained shortly when comparing disposal semantics in this setting.

### 4.2. Contiguous vs. Non-contiguous Matching

In addition to FR and MR matching, the added constraint that matched events must be contiguous may cause the violation of validity.

**THEOREM** 5. *Contiguous matching violates* EVENT VALIDITY *and* CONJUNCTION VALIDITY

PROOF. This proof will be by counter example. Suppose that a process $p_i$ has a subscription on a type $T_1$ such that $\Phi = T_1[1].a_1 = v \wedge T_1[2].a_1 = v$, which attempts to match two events from the same stream each with a first attribute with a value of $v$. In this scenario, it is possible that no two consecutive events have the same value for the first attribute. Suppose that a process sends events such that $a_1$ alternates between values $v$ and some $v'$ such that $v' \neq v$. Thus, with contiguous matching, no two consecutive events have the value $v$, whereas with non-contiguous matching, a match is possible by considering every other event. Thus both EVENT VALIDITY and CONJUNCTION VALIDITY are violated. □

Consider a simple subscription: $\Phi_U = \texttt{SQuote}[0].\texttt{value} = 3.44 \wedge \texttt{SQuote}[1].\texttt{value} = 3.44$ which looks for two stock quote events to have the same value. It is easy to see that if published events of type `SQuote` are consistently alternating between different values between any two events, then with the requirement of contiguous matching, there would never be a match for $\Phi_U$, thus violating EVENT VALIDITY and CONJUNCTION VALIDITY since no events would ever be delivered. This could be solved by matching stock quote events in a non-contiguous manner.

### 4.3. Infix vs. Prefix+Infix vs. Infix+Postfix Event Consumption

Taken by itself, PI disposal does not violate any properties as shown by the left middle portion of Table I. The properties violated with PI disposal together with MR matching are due to the matching semantics as shown in Section 4.1.1. In contrast though, I and IP disposal cause the violation of a number of properties.

*4.3.1. Properties of Infix Only Event Consumption.* I disposal causes the violation of the properties TYPE TOTAL ORDER, TYPE FIFO ORDER and TYPE CAUSAL ORDER. This is due to the fact of how different events may be correlated over time. The following theorem demonstrates why I disposal can violate all three properties simultaneously.

**THEOREM** 6. *I disposal violates* TYPE TOTAL ORDER*,* TYPE FIFO ORDER *and* TYPE CAUSAL ORDER

PROOF. By counter example, consider two processes $p_i$ and $p_j$ such that their subscriptions are $\Phi_i = T_1.a_1 < T_2.a_1$ and $\Phi_j = T_1.a_1 > T_3.a_1$ respectively. Now suppose that both $p_i$ and $p_j$ (starting with empty queues) receive the event $e_1^1$ of type $T_1$ such that $e_1^1.a_1 = v$. Since this is the only event which either process has received, then both will queue $e_1^1$ for later matching. Now, suppose that $p_j$ receives the event $e_1^3$ of type $T_3$ such that $e_1^3.a_1 = w$ (s.t. $v > w$). Now, process $p_j$ may trigger a match and deliver the relation $[e_1^1, e_1^3]$. This match would be triggered by either MR or FR matching. Next, suppose that both $p_i$ and $p_j$ receive another event $e_2^1$ of type $T_1$ such that $e_2^1.a_1 = v'$ and then $p_j$ receives an event $e_2^3$ of type $T_3$ such that $e_2^3.a_1 = w'$ (s.t. $v' > w'$). The process $p_j$ may now trigger another match and deliver the relation $[e_2^1, e_2^3]$, which may be matched by either MR or FR matching. Since $p_i$ has not yet received any events of type $T_2$, it may not yet deliver any relations.

Note, at this point, no properties have yet been violated. Now suppose that process $p_i$ receives an event $e_1^2$ of type $T_2$ such that $e_1^2.a_1 = u$ (s.t. $v' < u < v$). By either MR or FR matching, when $p_i$ attempts to trigger a match, $e_1^2$ will only match with $e_2^1$ and thus $p_i$ delivers the relation $[e_2^1, e_1^2]$. By I disposal, $p_i$ discards only the events delivered, and thus the event $e_1^1$ remains in $p_i$'s queue. Again, at this moment, no properties have yet been violated. But if $p_i$ were to now receive an event $e_2^2$ of type $T_2$ such that $e_2^2.a_1 = u'$ (s.t. $v < u'$), this event may be matched by $p_i$ with $e_1^1$ and $p_i$ would thus deliver the relation $[e_1^1, e_2^2]$.

TYPE TOTAL ORDER has been violated since both $p_i$ and $p_j$ have different conjunctions, but receive events over a common type, i.e., $T_1$. Since $p_j$ delivers $e_1^1$ *before* $e_2^1$ within separate relations, but $p_i$ delivers $e_2^1$ before $e_1^1$ within separate relations, this total order over the events of the same common type $T_1$ is thus violated.

The above also shows the violation of TYPE FIFO ORDER since the event $e_1^1$ was clearly sent before $e_2^1$ (it may be assumed that both were sent by the same process for the sake of argument), but $p_i$ delivered those events in a conflicting order. Lastly, since causal order requires FIFO order and FIFO order has here been violated, it follows that TYPE CAUSAL ORDER may also be violated. □

As an example, consider the two subscriptions (by processes $p_1$ and $p_2$ respectively) that resemble the subscriptions in the counter example above:

$\Phi_{V_1}$ = SQuote[0].value < Euro[0].value
$\Phi_{V_2}$ = SQuote[0].value < USDollar[0].value

As in the counter example, consider the following events are received in the following order (where Type($v$) represents receiving an event of type Type with value $v$): {SQuote(3), USDollar(1), SQuote(2), USDollar(0.98)} In this case, $p_2$ may deliver the relations [SQuote(3), USDollar(1)] and [SQuote(2), USDollar(0.98)] by $\Phi_{V_2}$ but $p_1$ has not yet received any events of type Euro yet, so both SQuote(3) and SQuote(2) are queued in that order. Now supposed the event Euro(2.5) is now received by $p_1$. The only possible relation that may be delivered by $p_1$ (using any matching semantics for $\Phi_{V_1}$) is thus [SQuote(2), Euro(2.5)] while the event SQuote(3) remains in $p_1$'s queue due to I disposal. However, if the event Euro(3.1) were then received by $p_1$, the relation [SQuote(3), Euro(3.1)] may then be delivered by $p_1$. In this case, it is clear that TYPE FIFO ORDER is violated since Euro(2.5) was sent and received before Euro(3.1) which also shows that TYPE TOTAL ORDER is violated over the type SQuote since $p_1$ and $p_2$ delivered the events of type SQuote in differing orders. Since TYPE FIFO ORDER is violated, TYPE CAUSAL ORDER is thus also violated.

*4.3.2. Properties of Infix+Postfix Event Consumption.* Section 4.1.1 discussed violation of EVENT VALIDITY and COVERING AGREEMENT by IP disposal with MR matching. These properties remain to be investigated in the context of FR matching.

**THEOREM** 7. *FR matching with IP disposal violates* EVENT VALIDITY

PROOF. By counter example, consider a process $p_i$ that has a subscription such that all predicates over a type $T_x$ are unary predicates, i.e., only comparing attributes of the type to scalar values. If $p_i$ were to start with an empty set of queues, and immediately received two events of type $T_x$ that both meet all the unary predicates over $T_x$ and then received other events which completed a match, the first event of type $T_x$ would be matched in a relation with the other received events and then the second would be discarded by IP disposal, thus violating EVENT VALIDITY. □

As an example, consider the subscription:
$\Phi_W = \texttt{SQuote}[0].\texttt{value} < 3 \wedge \texttt{USDollar}[0].\texttt{value} < 1$
If a process with the subscription $\Phi_W$ receives the two events SQuote(2.5) and SQuote(2) respectively, no relations may yet be delivered. However, if an event USDollar(0.98) were received, then by FR matching, the relation [SQuote(2.5), USDollar(0.98)] may then be delivered. By using IP disposal, then the event SQuote(2) is discarded, which violates EVENT VALIDITY.

**THEOREM** 8. *FR matching with IP disposal violates* COVERING AGREEMENT

PROOF. Consider, again by counter-example, two processes $p_i$ and $p_j$ with subscriptions $\Phi_i = T_1.a_1 < T_2.a_1$ and $\Phi_j = \Phi_i \wedge T_3.a_1 < v$ respectively. If both $p_i$ and $p_j$, starting with empty queues, receive two events $e_1^1$ and $e_1^2$ such that $e_1^1.a_1 = u$ and $e_1^2.a_1 = u'$ (s.t. $u < u'$), then $p_i$ may deliver the relation $[e_1^1, e_1^2]$ whereas $p_j$ must wait for a matching event of type $T_3$. Next, if both $p_i$ and $p_j$ were to receive two more events $e_2^1$ and $e_2^2$ such that $e_2^1.a_1 = u''$ and $e_2^2.a_1 = u'''$ (s.t. $u'' < u'''$), again $p_i$ may deliver another relation $[e_2^1, e_2^2]$, but $p_j$ must wait for a matching event of type $T_3$. Lastly, if $p_j$ were to then receive an event $e_1^3$ such that $e_1^3.a_1 = w$ (s.t. $w < v$), using FR matching, $p_j$ may now perform a match and deliver the relation $[e_1^1, e_1^2, e_1^3]$. However, due to IP disposal, $p_j$ will discard both $e_2^1$ and $e_2^2$ since these are in the same type queues as the delivered events of types $T_1$ and $T_2$. Thus, COVERING AGREEMENT is violated since $p_i$ delivered the two events $e_2^1$ and $e_2^2$ whereas $p_j$ discarded them. □

Consider the subscriptions for processes $p_1$ and $p_2$ respectively:
$\Phi_{X_1} = \texttt{SQuote}[0].\texttt{value} < \texttt{USDollar}[0].\texttt{value}$
$\Phi_{X_2} = \texttt{SQuote}[0].\texttt{value} < \texttt{USDollar}[0].\texttt{value} \wedge \texttt{USDollar}[0].\texttt{value} < 1.9$
If both $p_1$ and $p_2$ receive the events SQuote(1) and USDollar(1.1) respectively, then only $p_1$ can deliver these events as a relation, whereas $p_2$ must place them in a queue to await an event of type Euro. If now, both processes receive the two events SQuote(1.2) and USDollar(1.3) respectively, then again, $p_1$ may deliver these events in a relation but $p_2$ cannot since there is still yet no event of type Euro with which to match these received events (i.e., the queue for the type Euro for $p_2$ is empty). If $p_2$ were to then receive the event Euro(1.8), then $p_2$ may now perform a match on $\Phi_{X_2}$. By FR matching, $p_2$ will deliver the relation [SQuote(1), USDollar(1.1), USDollar(1.8)]. However, by IP disposal, $p_2$ will then discard the events SQuote(1.2) and USDollar(1.3) from its queue. Since $p_2$ will thus never deliver these discarded events, $p_1$ and $p_2$ will not agree on all sub-relations delivered over the types SQuote and USDollar, and thus COVERING AGREEMENT is violated.

### 4.4. Tumbling vs. Sliding Windows

Windows in this context are equivalent to replacing the disposal semantics. In particular, note that tumbling windows are equivalent to using PI disposal. Thus, what remains to be discussed is the topic of sliding windows. Sliding windows may be implemented as either a *contiguous* sliding window, i.e., all events matched within the same window must be contiguous, or a *non-contiguous* sliding window where as long as all events matched are currently in the window, they need not be contiguous. Note that for contiguous sliding windows, the only additional property that is unmet, aside from those by the matching semantics, is NO DUPLICATION; however, NO DUPLICATION′ may still be met. Non-contiguous sliding windows also violate NO DUPLICATION while maintaining NO DUPLICATION′.

*4.4.1. Properties of Sliding Windows.* Sliding windows can violate NO DUPLICATION as shown below in demonstrating that FP violates this property.

**THEOREM** 9. *FP disposal violates* NO DUPLICATION.

PROOF. By counter-example, suppose that a process $p_i$ has a conjunction $\Phi = T_1[1].a_1 < T_1[2].a_1$. Now, suppose that $p_i$ receives three events $e_1^1$, $e_2^1$ and $e_3^1$ of type $T_1$ such that $e_1^1.a_1 = v$, $e_2^1.a_1 = v'$ and $e_3^1.a_1 = v''$ (s.t. $v < v' < v''$). Process $p_i$ will first deliver the relation (here by FR) $\{e_1^1, e_2^1\}$ and then discard $e_1^1$ by FP disposal. Process $p_i$ will then deliver the relation $\{e_2^1, e_3^1\}$ also by FR. Since $e_2^1$ is delivered within more than one relation, NO DUPLICATION is violated. This same argument holds for MR matching, with the relations delivered thus being □

As a quick example, consider the subscription $\Phi_Y =$ `SQuote`$[0]$.`value` $<$ `SQuote`$[1]$.`value` $\wedge$ `USDollar`$[0]$.`value` $< 1.1$ Suppose that a process with subscription $\Phi_Y$ received (in the following order) the events `SQuote`(1), `SQuote`(1.1) and `SQuote`(1.2) before receiving any events of type `USDollar`, thus no relations are yet able to be delivered. If an event `USDollar`(1) were then received, then the process will (by FR matching) deliver the relation [`SQuote`(1), `SQuote`(1.1), `USDollar`(1)] and by FP disposal, the only events to be discarded from each of the queues respectively are `SQuote`(1) and `USDollar`(1). This leaves the queue for the `SQuote` with the two events `SQuote`(1.1) (which has been delivered as part of a relation), and `SQuote`(1.2) respectfully with the queue for `USDollar` now empty. Lastly, if an event `USDollar`(0.99) were received, then another relation may be delivered. By FR matching, the relation delivered is thus [`SQuote`(1.1), `SQuote`(1.2), `USDollar`(0.99)] with the only events discarded are thus `SQuote`(1.1) and `USDollar`(0.99) (by FP, leaving `SQuote`(1.2) in the queue. Since between the two relations [`SQuote`(1), `SQuote`(1.1), `USDollar`(1)] and [`SQuote`(1.1), `SQuote`(1.2), `USDollar`(0.99)], one can see that `SQuote`(1.1) was delivered in two separate relations, which thus violates NO DUPLICATION.

Note that in the proof above, since $e_2^1$ (in the counter-example) is delivered in different positions between the two relations, NO DUPLICATION$'$ is retained in both the proof counter-example and the provided example following the proof.

Sliding windows with non-contiguous matching may cause the violation of a number of the ordering properties. The following theorem demonstrates how non-contiguous matching with a sliding window via FP disposal may cause the violation of the properties TYPE TOTAL ORDER, TYPE FIFO ORDER, and TYPE CAUSAL ORDER.

**THEOREM** 10. *Both (non-contiguous) FR and MR matching with FP disposal violate* TYPE FIFO ORDER, *and* TYPE CAUSAL ORDER

PROOF. The following is by counter-example. Consider a subscription by process $p_i$, $\Phi = T_1[1].a_1 = T_1[2].a_1$ where this subscription denotes that an attribute of some event of type $T_1$ must be equal to the same attribute of a later received event. Again, in this semantic, these are not guaranteed to be contiguous events. Consider if a current queue for process $p_i$ were $[e_1^1, e_2^1, e_3^1, e_4^1]$ where $a_1$ for each of these events are respectively $[v, v', v, v']$. Consider FR matching. The first relation to be delivered would thus be $\{e_1^1, e_3^1\}$. By FP disposal, the event $e_1^1$ would be discarded. The next delivered relation would then be $\{e_2^1, e_4^1\}$. In this case, since $e_2^1$ is delivered *after* $e_3^1$, the aforementioned ordering properties are thus violated. This same argument can be used for MR matching. □

Consider a subscription $\Phi_Z =$ `SQuote`$[0]$.`value` $=$ `SQuote`$[1]$.`value`. Suppose that the incoming quotes alternated values, such that a process with subscription $\Phi_Z$ receives the following events in order: `SQuote`(1.1), `SQuote`(1.2) and `SQuote`(1.1). By the subscription, the process would deliver the relation [`SQuote`(1.1), `SQuote`(1.1)] (the first and third received events). If using FP disposal, then only the first event would be discarded, with the resulting queue being $\{$`SQuote`(1.2), `SQuote`(1.1)$\}$. Now suppose a fourth event `SQuote`(1.2) were received. The relation [`SQuote`(1.2), `SQuote`(1.2)] may then be delivered. However, in this case, since the first instance of `SQuote`(1.2) was received *before* the second instance of `SQuote`(1.1), but they were delivered such that the second instance of `SQuote`(1.1) was delivered first, then Type FIFO Order is violated.

## 5. CASE STUDIES

In this section we investigate the properties obtained when substituting previously proposed correlation engines/languages in the FAIDECS overlay network. We investigate how their constructs relate to the properties previously introduced by mapping them to the semantic options discussed. Tables II and III summarize our findings. TESLA is discussed in the following; StreamSQL, CEL and EQL are presented in our technical report [Wilkin et al. 2014].

| | Basic Safety | | | | Liveness | |
|---|---|---|---|---|---|---|
| | No Duplication | No Duplication′ | No Creation | Admission | Conjunction Validity | Event Validity |
| TESLA | `each-within` | - | - | - | - | `first-within`<br>`last-within`<br>`not` |
| StreamSQL | `select` | - | - | - | `select`<br>`create window` | `select` |
| EQL | `select` | - | - | - | `select`<br>`create window` | `select`<br>`limit` |
| CEL | `select` | - | - | - | `select` | `select` |

Table II: Basic safety as well as liveness properties violated by various language operators.

| | Agreement | | Order | | | |
|---|---|---|---|---|---|---|
| | Conjunction Agreement | Covering Agreement | Type Total Order | Conjunction Total Order | FIFO Order | Causal Order |
| TESLA | - | `first-within`<br>`last-within` | `each-within`<br>`consuming` | - | `each-within`<br>`consuming` | `each-within`<br>`consuming` |
| StreamSQL | - | `create window` | `select`, `union`,<br>`merge` | - | `select`, `union`<br>`create window`<br>`merge` | `select`, `union`<br>`create window`<br>`merge` |
| EQL | - | `create window` | `select`, `union`,<br>`merge` | - | `select`, `union`<br>`create window`<br>`merge` | `select`, `union`<br>`create window`<br>`merge` |
| CEL | - | - | `select` | - | `select` | `select` |

Table III: Agreement and ordering (safety) properties violated by various language operators.

### 5.1. The TESLA Language

TESLA [Cugola and Margara 2010], a complex event specification language, provides a high degree of expressiveness and flexibility for event subscriptions with an intuitive and simple syntax. In particular, the operators that TESLA provides are operators for event occurrence, event composition, parameterization, timers, negation, event consumption, aggregates, event hierarchies and iterations. The following represents a general TESLA query.

```
define subscription([att_1 : type_1,  ..., att_n : type_n])
  from event_source ([pattern]) [and interval_operation]
  [where predicate] [consuming event_identifiers]
```

Replacing the matching logic (i.e., the matching and disposal semantics) of FAIDECS with that of TESLA would thus allow for a much more expressive event correlation system. The following describes each of the operators of TESLA, and how the addition of each to FAIDECS affects the respective properties.

*5.1.1. Event Occurrence/Selection.* TESLA allows for a simple subscription, specifying constraints over singleton events in both content and time. Because the properties of FAIDECS may be simplified for single event delivery, all aforementioned properties still hold for these operators. The following is in the syntax of TESLA using the semantics of FAIDECS to denote events and their types and attributes:

```
define Subscription1() from SQuote (SQuote.val > 10)
```

The equivalent subscription in FAIDECS is $\Phi = \texttt{SQuote}[0].\texttt{val} > 10$.

*5.1.2. Event Composition.* Event correlation is possible in TESLA through event composition operators. TESLA provides three variants with specific matching and disposal rules associate with each. They are, respectively, **each-within**, **first-within** and **last-within**. The idea regarding these operators is that in specifying an event composition, it is possible that a single event could be matched with one or more events to make composite events or relations. In particular, when events are to be matched within a certain time interval of the occurrence of some singular event, these operators specify *precisely* how the single event is to be correlated with the others.

The **each-within** operator provides the most composite events. This operator is equivalent to FR matching with I disposal of Table I. An example subscription in TESLA follows:

```
define Subscription2() from EReport() and
  each SQuote(SQuote.val > 10) within 5min from EReport
```

In this subscription, any occurrence of an event of type `EReport` would be saved for five minutes to be matched with any of type `SQuote` where `SQuote.val` $> 10$. For events of type `EReport`, the property NO DUPLICATION is not met, but as discussed for windows, NO DUPLICATION$'$ may be met instead. For events of type `SQuote`, all events are matched in a FR order and I disposal applies. Thus, by Table I, the properties TYPE TOTAL ORDER, TYPE FIFO ORDER and TYPE CAUSAL ORDER are violated. Thus, if an event $e^{ER}$ of type `EReport` were received, any and all events of type `SQuote` for which `SQuote.val` $> 10$, received within five minutes after having received $e^{ER}$, will be delivered in separate relations with $e^{ER}$.

The **first-within** operator only allows for a single composite event or relation to be delivered for a given subscription within the specified time interval. In the above subscription of Section 5.1.2, if replacing the keyword **each** with **first**, then of all events received of type $T_2$ within five minutes of receiving an event $e^1$ of type $T_1$, only the *first* event for which $e^2.a > 10$ will be delivered.

Depending on *when* the matching is triggered, in the worst case, the **first-within** operator is equivalent to FR matching with PI disposal for all events of type $T_2$. Thus, by Table I, the properties that are violated are EVENT VALIDITY and COVERING AGREEMENT.

The **last-within** operator, similar to **first-within**, allows only for a single composite event or relation to be delivered within a specific time interval. By replacing **each** with **last** in the example subscription in Section 5.1.2, then of all events received of type $T_2$ within five minutes of receiving an event $e^1$ of type $T_1$, only the *last* event for which $e^2.a > 10$ will be delivered.

The properties which are violated again depend on when the matching is triggered, but in the worst case, the **last-within** operator is equivalent to most-recently received matching with PI disposal. By Table I, this operator thus violates EVENT VALIDITY and COVERING AGREEMENT.

*5.1.3. Parameterization.* Parameterization in the context of TESLA is the composition of events when related by some higher order function such as **area**. An example of a parameterized subscription, in English, could thus be: *Warn of an avalanche when 3 or more sensors detect movement when these sensors are within the same area $x*. Where *area $x* can be specified as a parameter in the subscription. In this case, location, or whatever other parameters, may be included within events as further attributes, which equates to nothing more than further constraints on attributes of events. Parameterization does not cause the violation of any of the above properties.

*5.1.4. Timers.* The TESLA language allows for events to be matched using timers. An example would be to attempt to trigger a match every morning at 10 a.m. over all received events since the last time the matching was triggered. Because this type of matching can use any matching and disposal semantics, this operator will not suffer further violations of properties aside from any that may be violated by the matching and disposal semantics themselves.

Note that the use of timers assumes at least a partially synchronous system, however, which is opposed to the assumptions of FAIDECS. However, specialized solutions do exist, which deal with such cases and are out of the scope of this paper.

*5.1.5. Negations.* The negation operator allows the control of when certain composite events should **`not`** be matched. Since this operator only specifies that certain events should not be delivered, only EVENT VALIDITY may be violated in this case.

*5.1.6. Aggregates.* Operators such as **`min`**, **`max`**, **`average`**, **`sum`**, etc., are examples of aggregate operators. Aggregate operators take more than one event from a particular queue and yield a single result. This is equivalent to consuming events in streams using a tumbling window, thus aggregates do not violate any properties.

*5.1.7. Event Consumption.* TESLA provides the expressiveness to specify which events should be consumed or discarded. Consider the following example:

```
define Subscription2() from EReport() and
  each SQuote(SQuote.val > 10) within 5min from EReport
  consuming SQuote
```

This subscription provides a specific disposal policy for all events of type `SQuote`. To avoid the scenario where the same events of type `SQuote` may be matched with multiple events of type `EReport`, the **`consuming`** keyword specifies that any events of that type may only be matched once, and then discarded such that any new events of type `EReport` must be matched with new events of type `SQuote`. This is equivalent to I disposal. Thus, the properties which may be violated will be the intersection between those violated by the composition operators (as specified in Section 5.1.2), and then the resulting matching semantics of those operators together with I disposal. The unmet properties are thus shown in Table I.

*5.1.8. Event Hierarchies.* Certain single events may be matched together to form a composite event or relation, which may be matched together to form further, more complex composite events comprised of simple events and complex events. These subscriptions require levels (i.e., hierarchies) of correlation. Hierarchies allow for more expressiveness while meeting all properties.

*5.1.9. Iterations.* Iterations specify constraints over a set of events of the same type over time. An example would be to "capture" every iteration of events of type $T_1$ such that the attribute $a_1$ never decreases. Due to TESLA's ability to define hierarchies of events and the ability to specify different selection and consumption policies for different rules, no further operators are needed to allow for iterations. In this case, again, when an iteration is specified, the violated properties are thus the intersection of the violated properties of the selection and consumption policies.

## 6. EVALUATION

To demonstrate the scalability of our decentralized algorithms and explore overall performance benefits and tradeoffs, we compare the performance of the FAICECS system using two different matching engines implemented in Java — Esper (http://esper.codehaus.org) and Jess (http://www.jessrules.com) — with three different implementations of a global total order: two fault tolerant ones and a non-replicated sequencer (with Esper and Jess again) for event correlation at the subscribers. We have included the non-replicated non-fault tolerant sequencer because that is the most efficient sequencer.

### 6.1. Metrics and Setup

We used two metrics: (1) *throughput* measures the average number of events delivered per second by a subscriber; (2) *latency* measures the average delay between the production time of an event and its delivery to a subscriber. We chose subscriptions based on the default workload in the Marketcetera algorithmic trading system (http://marketcetera.org/). In this workload, the publisher is the Marketcetera stock exchange simulator, and the subscribers are algorithmic traders. The default workload has 23 event types, and several conjunctions. The maximum number of event types in any conjunction is 6. The number of subscribers (traders) was increased from 10 to 500. We used three nodes for Paxos and the token passing total order implementation, i.e., the state of the replicated fault tolerant sequencer was replicated on three nodes. For both Paxos and Token-passing total order, the publisher sent its events randomly to one of the three nodes.

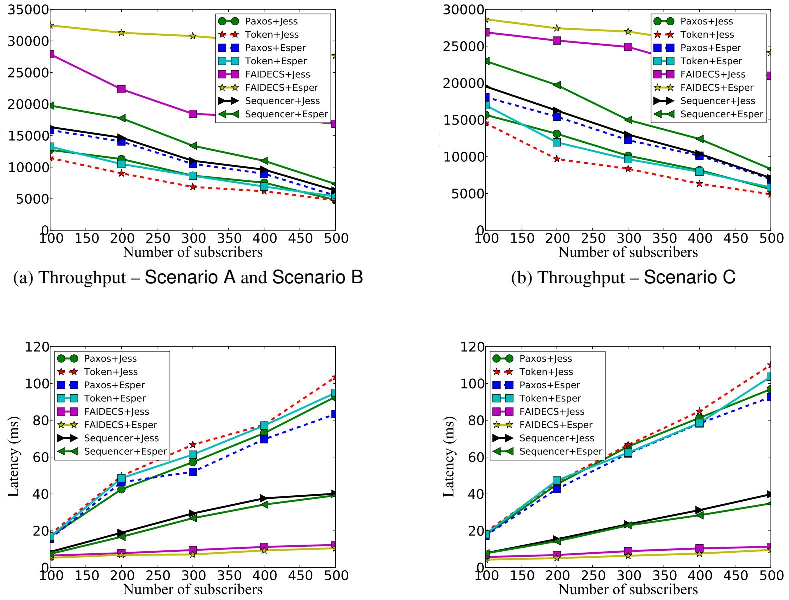
We have three deployment scenarios. With FAIDECS, conjunctions are performed by merger processes and predicates are evaluated at the subscribers by two popular event correlation systems – Jess (originally used in [Wilkin et al. 2011]) and Esper. In Scenario A and Scenario B, we used a setup for conjunctions similar to Figure 2. All filtering occurred at end nodes rather than in mergers through the selectivity of binary predicates, which differed across conjunctions to achieve the same expected delivery rates at all subscribers in a respective level. This scenario demonstrated the limits of the overlay. In Scenario B, events were filtered at the mergers through unary predicates propagated upwards from subscriptions, allowing higher aggregate multicast rates than in Scenario A. In Scenario C, we statistically generated subscriptions uniformly over all event types in the system with all possible conjunction combinations. This allowed us to explore the potential of traffic separation. Subscribers were uniformly distributed across all merger processes and throughput/latency values were averaged for each group of subscribers for a given level. We expect that the bottleneck in our decentralized algorithms would occur at the merger process(es), which would merge all involved types, limiting throughput consistently for all overlay depths from either the publisher or subscriber.

### 6.2. Results

Figure 9 illustrates our results, both for throughput and latency. We observe from all four figures that the results for the sequencer are better than possibly expected. This is because the sequencer we used was a non-fault tolerant counter. Regardless, both versions of FAIDECS easily outperform the corresponding sequencer implementations. This demonstrates how correlation-specific ordering enables strong guarantees even with support for fault tolerance. Both Jess and Esper are current state of the art correlation languages, but in some scenarios, as seen above, Esper is more efficient than Jess since Esper uses a more optimized event correlation algorithm. In either case however, the benefits of the FAIDECS overlay are preserved – in fact, a more efficient matching engine further amplifies its benefits. For Scenario A and Scenario B, the throughput of FAIDECS is at least 84%-4.82× higher than that of the sequencer. The corresponding numbers for Scenario C are 59% to 4.12× higher than that of the sequencer. The difference in latency is yet more pronounced, because lower throughput typically has a cascading effect on latency when the number of subscribers is high. The latency of FAIDECS is up to 3.7× lower than the sequencer, and scales much better than the sequencer. The throughput of the implementations with Paxos and Token-passing total order are lower than Sequencer, though sometimes the differences are less pronounced due to the processing in Esper and Jess at the subscribers.

## 7. RELATED WORK

Event correlation has been vigorously investigated in the context of *content-based publish/subscribe systems*. Most such systems rely on a *broker network* for routing events to the subscribers (e.g., SIENA [Carzaniga et al. 2001] and Gryphon [Aguilera et al. 1999]). *Advertisements* are typically used to form routing trees in order to avoid propagating subscriptions by flooding the broker network. Upon receiving an event $e$, a broker determines the subset of parties (subscribers and brokers)



(a) Throughput – Scenario A and Scenario B

(b) Throughput – Scenario C

(c) Latency – Scenario A and Scenario B

(d) Latency – Scenario C

Fig. 9: Empirical evaluation of FAIDECS

with matching interests and forwards $e$ to them. Subscription *subsumption* [Carzaniga et al. 2001] is used to summarize subscriptions and avoid redundant matching on brokers and redundant traffic among them. If any event $e$ that matches a first subscription also matches a second one, then the latter subscription subsumes the former one.

A broker network can be used to gather all publications for the elementary subscriptions and perform correlation matching. A successful match yields a composite event which is delivered to interested subscribers, where no guarantees are typically provided on correlation. If the events matching a composite subscription shared by two subscribers are produced by several publishers, then unless the subscribers are connected to a same edge broker, they may receive the events through different routes. This leads to different orders among the events and consequently to different composite events for the two subscribers. PADRES [Li and Jacobsen 2005] performs composite event detection for each subscription at the first broker that accumulates all the individual subscriptions, providing no global properties. Hermes [Pietzuch et al. 2003] proposes *complex event detectors* using an interval timestamp model as a generic extension for existing middleware architectures. Hermes uses a DHT to determine rendezvous nodes for publishers and subscribers; however, these are not replicated for fault-tolerance.

Recent work [Zhang et al. 2012], motivated by solving agreed correlation, further demonstrates the need for stronger guarantees on correlated deliveries. However the approach proposes a more generic primitive for publish/subscribe systems which is opportunistically layered on top of an existing overlay network, leading to high overhead.

Hummer et al. [Hummer et al. 2012] propose a unified fault taxonomy for general event-based systems. This work generically categorizes faults into separate classes as well as the sources for these faults to better detect and predict faults in future systems.

The work of Lumezanu et al. [Lumezanu et al. 2006] proposes a decentralized network of sequencers and uses a DHT for load balancing. However, this work only provides total order among messages of the same type/topic, and not for conjunctions, and thus differs from FAIDECS, which performs decentralized merging for conjunctions of types. One could implement FAIDECS-style mergers on top of Lumezanu et al.'s work [Lumezanu et al. 2006] by mapping conjunctions as types in the DHT and routing messages from the node responsible for a type to a node responsible for a conjunction. (The merging additionally would take predicates into account.) A similar approach could be used to deal with disjunctions (omitted from this paper for simplicity). The work of Baldoni et al. [Baldoni et al. 2012] establishes an ordering among topics, and totally orders events within topics and to some degree across, however without distinguishing (guarantees) across conjunctions and disjunctions. Their system is devised to work on top of an arbitrary basic publish/subscribe system (which improves its portability but adversely affects latency), but then still allows messages to be explicitly delivered out of order to the application with a corresponding specific notification.

TimeStream [Qian et al. 2013] is a recent fault-tolerant stream processing architecture, which is similar to Apache Storm, except for additional reconfiguration and re-starting guarantees provided to stream processing elements. However, TimeStream does not provide ordering guarantees because it is targeted at generic stream processing, where each processing element contains arbitrary code, and is not targeted at events or tuples of data. Aurora [Abadi et al. 2003] and its successor Borealis [Balazinska et al. 2008] are seminal stream processing systems, where Borealis uses replication for fault tolerance. Each replica processes events in the same order, and Borealis provides TYPE TOTAL ORDER, but does not provide CONJUNCTION TOTAL ORDER, i.e., in Borealis, it is possible to obtain total order among subscribers for all messages delivered on a given type, e.g., "StockQuote", but not a total order among messages delivered to subscribers on a join or a conjunction, e.g., "StockQuote and AnalystReport". The guarantees provided by System S [Jacques-Silva et al. 2007] are similar to Borealis, but the mechanisms (E.g. checkpointing techniques and orchestration) differ. Cayuga [Demers et al. 2006] is a generic correlation engine supporting correlation across streams and is based on a very expressive language but is centralized.

## 8. CONCLUSIONS

FAIDECS presents a powerful event correlation model for trading between (a) strong guarantees in the face of failures and (b) performance; its implementation hinges on an overlay network for deterministic type-wise merging of event flows with replication of merger nodes. We have presented semantic options for several modules of the FAIDECS matching engine. We have shown for each of these alternatives which of the proposed properties are maintained and which are violated. We have investigated four correlation languages – StreamSQL, EQL, CEL and TESLA – and have mapped their features to the semantic options introduced. This then determines which properties are withheld when replacing the matching engine of FAIDECS with that of the respective correlation languages. To demonstrate that the benefits of the FAIDECS overlay in terms of performance (besides fault tolerance) are not dependent on any specific matching engine (while its specific properties do depend on the corresponding correlation language) we substituted the default engine of FAIDECS (Jess) by the more efficient Esper engine. As we have illustrated, Esper in fact amplifies the benefits of the FAIDECS overlay. We are currently investigating further semantic options in the context of *disjunctions* and other security features for FAIDECS.

## REFERENCES

[Abadi et al. 2003] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. 2003. Aurora: A New Model and Architecture for Data Stream Management. *The VLDB Journal* 12, 2 (Aug. 2003), 120–139. DOI:http://dx.doi.org/10.1007/s00778-003-0095-z

[Aguilera et al. 1999] Marcos K. Aguilera, Robert E. Strom, Daniel C. Sturman, Mark Astley, and Tushar D. Chandra. 1999. Matching Events in a Content-based Subscription System. In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing (PODC '99)*. ACM, New York, NY, USA, 53–61. DOI:http://dx.doi.org/10.1145/301308.301326

[Aguilera and Toueg 1996] Marcos Kawazoe Aguilera and Sam Toueg. 1996. Randomization and Failure Detection: A Hybrid Approach to Solve Consensus. In *Proceedings of the 10th International Workshop on Distributed Algorithms (WDAG '96)*. Springer-Verlag, London, UK, 29–39. http://dl.acm.org/citation.cfm?id=645953.675629

[Balazinska et al. 2008] Magdalena Balazinska, Hari Balakrishnan, Samuel R. Madden, and Michael Stonebraker. 2008. Fault-tolerance in the Borealis Distributed Stream Processing System. *ACM Trans. Database Syst.* 33, 1, Article 3 (March 2008), 44 pages. DOI:http://dx.doi.org/10.1145/1331904.1331907

[Baldoni et al. 2012] Roberto Baldoni, Silvia Bonomi, Marco Platania, and Leonardo Querzoni. 2012. Dynamic Message Ordering for Topic-Based Publish/Subscribe Systems. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS '12)*. IEEE Computer Society, Washington, DC, USA, 909–920. DOI:http://dx.doi.org/10.1109/IPDPS.2012.86

[Basu et al. 1996] Anindya Basu, Bernadette Charron-Bost, and Sam Toueg. 1996. Simulating Reliable Links with Unreliable Links in the Presence of Process Crashes. In *Proceedings of the 10th International Workshop on Distributed Algorithms (WDAG '96)*. Springer-Verlag, London, UK, 105–122. http://dl.acm.org/citation.cfm?id=645953.675641

[Brenna et al. 2007] Lars Brenna, Alan Demers, Johannes Gehrke, Mingsheng Hong, Joel Ossher, Biswanath Panda, Mirek Riedewald, Mohit Thatte, and Walker White. 2007. Cayuga: A High-performance Event Processing Engine. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data (SIGMOD '07)*. ACM, New York, NY, USA, 1100–1102. DOI:http://dx.doi.org/10.1145/1247480.1247620

[Carzaniga et al. 2001] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. 2001. Design and Evaluation of a Wide-area Event Notification Service. *ACM Trans. Comput. Syst.* 19, 3 (Aug. 2001), 332–383. DOI:http://dx.doi.org/10.1145/380749.380767

[Chakravarthy et al. 1994] Sharma Chakravarthy, V. Krishnaprasad, Eman Anwar, and S.-K. Kim. 1994. Composite Events for Active Databases: Semantics, Contexts and Detection. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB '94)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 606–617. http://dl.acm.org/citation.cfm?id=645920.672994

[Chandra and Toueg 1996] Tushar Deepak Chandra and Sam Toueg. 1996. Unreliable Failure Detectors for Reliable Distributed Systems. *J. ACM* 43, 2 (March 1996), 225–267. DOI:http://dx.doi.org/10.1145/226643.226647

[Cugola and Margara 2010] Gianpaolo Cugola and Alessandro Margara. 2010. TESLA: A Formally Defined Event Specification Language. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems (DEBS '10)*. ACM, New York, NY, USA, 50–61. DOI:http://dx.doi.org/10.1145/1827418.1827427

[Défago et al. 2004] Xavier Défago, André Schiper, and Péter Urbán. 2004. Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey. *ACM Comput. Surv.* 36, 4 (Dec. 2004), 372–421. DOI:http://dx.doi.org/10.1145/1041680.1041682

[Demers et al. 2006] Alan Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker White. 2006. Towards Expressive Publish/Subscribe Systems. In *Proceedings of the 10th International Conference on Advances in Database Technology (EDBT'06)*. Springer-Verlag, Berlin, Heidelberg, 627–644. DOI:http://dx.doi.org/10.1007/11687238_38

[Fischer et al. 1985] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1985. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM* 32, 2 (April 1985), 374–382. DOI:http://dx.doi.org/10.1145/3149.214121

[Gatziu and Dittrich 1994] S. Gatziu and K.R. Dittrich. 1994. Detecting composite events in active database systems using Petri nets. In *Research Issues in Data Engineering, 1994. Active Database Systems. Proceedings Fourth International Workshop on*. 2–9. DOI:http://dx.doi.org/10.1109/RIDE.1994.282859

[Gehani et al. 1992] Narain H. Gehani, H. V. Jagadish, and Oded Shmueli. 1992. Composite Event Specification in Active Databases: Model &Amp; Implementation. In *Proceedings of the 18th International Conference on Very Large Data Bases (VLDB '92)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 327–338. http://dl.acm.org/citation.cfm?id=645918.672484

[Hadzilacos and Toueg 1993] Vassos Hadzilacos and Sam Toueg. 1993. Distributed Systems (2nd Ed.). ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, Chapter Fault-tolerant Broadcasts and Related Problems, 97–145. http://dl.acm.org/citation.cfm?id=302430.302435

[Hummer et al. 2012] Waldemar Hummer, Christian Inzinger, Philipp Leitner, Benjamin Satzger, and Schahram Dustdar. 2012. Deriving a Unified Fault Taxonomy for Event-based Systems. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems (DEBS '12)*. ACM, New York, NY, USA, 167–178. DOI:http://dx.doi.org/10.1145/2335484.2335504

[Jacques-Silva et al. 2007] Gabriela Jacques-Silva, Jim Challenger, Lou Degenaro, James Giles, and Rohit Wagle. 2007. Towards Autonomic Fault Recovery in System-S. In *Proceedings of the Fourth International Conference on Autonomic Computing (ICAC '07)*. IEEE Computer Society, Washington, DC, USA, 31–. DOI:http://dx.doi.org/10.1109/ICAC.2007.40

[Jain et al. 2008] Namit Jain, Shailendra Mishra, Anand Srinivasan, Johannes Gehrke, Jennifer Widom, Hari Balakrishnan, Uğur Çetintemel, Mitch Cherniack, Richard Tibbetts, and Stan Zdonik. 2008. Towards a Streaming SQL Standard. *Proc. VLDB Endow.* 1, 2 (Aug. 2008), 1379–1390. DOI:http://dx.doi.org/10.14778/1454159.1454179

[Kompella et al. 2005] Ramana Rao Kompella, Jennifer Yates, Albert Greenberg, and Alex C. Snoeren. 2005. IP Fault Localization via Risk Modeling. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2 (NSDI'05)*. USENIX Association, Berkeley, CA, USA, 57–70. http://dl.acm.org/citation.cfm?id=1251203.1251208

[Krügel et al. 2002] Christopher Krügel, Thomas Toth, and Clemens Kerer. 2002. Decentralized Event Correlation for Intrusion Detection. In *Proceedings of the 4th International Conference Seoul on Information Security and Cryptology (ICISC '01)*. Springer-Verlag, London, UK, 114–131. http://dl.acm.org/citation.cfm?id=646283.687988

[Li and Jacobsen 2005] Guoli Li and Hans-Arno Jacobsen. 2005. Composite Subscriptions in Content-based Publish/Subscribe Systems. In *Proceedings of the ACM/IFIP/USENIX 2005 International Conference on Middleware (Middleware '05)*. Springer-Verlag New York, Inc., New York, NY, USA, 249–269. http://dl.acm.org/citation.cfm?id=1515890.1515903

[Lumezanu et al. 2006] Cristian Lumezanu, Neil Spring, and Bobby Bhattacharjee. 2006. Decentralized Message Ordering for Publish/-Subscribe Systems. In *Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware (Middleware '06)*. Springer-Verlag New York, Inc., New York, NY, USA, 162–179. http://dl.acm.org/citation.cfm?id=1515984.1515997

[Pietzuch et al. 2003] Peter R. Pietzuch, Brian Shand, and Jean Bacon. 2003. A Framework for Event Composition in Distributed Systems. In *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware (Middleware '03)*. Springer-Verlag New York, Inc., New York, NY, USA, 62–82. http://dl.acm.org/citation.cfm?id=1515915.1515921

[Qian et al. 2013] Zhengping Qian, Yong He, Chunzhi Su, Zhuojie Wu, Hongyu Zhu, Taizhi Zhang, Lidong Zhou, Yuan Yu, and Zheng Zhang. 2013. TimeStream: Reliable Stream Computation in the Cloud. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13)*. ACM, New York, NY, USA, 1–14. DOI:http://dx.doi.org/10.1145/2465351.2465353

[Sturzrehm et al. 2009] Heiko Sturzrehm, Pascal Felber, and Christof Fetzer. 2009. TM-Stream: An STM framework for distributed event stream processing. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. 1–8. DOI:http://dx.doi.org/10.1109/IPDPS.2009.5161084

[Triantafillou and Economides 2004] P. Triantafillou and A. Economides. 2004. Subscription summarization: a new paradigm for efficient publish/subscribe systems. In *Distributed Computing Systems, 2004. Proceedings. 24th International Conference on*. 562–571. DOI:http://dx.doi.org/10.1109/ICDCS.2004.1281623

[Wilkin and Eugster 2013] Gregory Aaron Wilkin and Patrick Eugster. 2013. Multicasting in the Presence of Aggregated Deliveries. *J. Parallel Distrib. Comput.* 73, 4 (April 2013), 544–556. DOI:http://dx.doi.org/10.1016/j.jpdc.2012.12.004

[Wilkin et al. 2014] Gregory Aaron Wilkin, Patrick Eugster, and K. R. Jayaram. 2014. Decentralized Fault Tolerant Event-Correlation. In *Technical Report*. http://www.jayaramkr.com/files/FAIDECSTechReport.pdf

[Wilkin et al. 2011] Gregory Aaron Wilkin, K. R. Jayaram, Patrick Eugster, and Ankur Khetrapal. 2011. FAIDECS: Fair Decentralized Event Correlation. In *Proceedings of the 12th ACM/IFIP/USENIX International Conference on Middleware (Middleware'11)*. Springer-Verlag, Berlin, Heidelberg, 228–248. DOI:http://dx.doi.org/10.1007/978-3-642-25821-3_12

[Zhang et al. 2012] Kaiwen Zhang, Vinod Muthusamy, and Hans-Arno Jacobsen. 2012. Total Order in Content-Based Publish/Subscribe Systems. In *Distributed Computing Systems (ICDCS), 2012 IEEE 32nd International Conference on*. 335–344. DOI:http://dx.doi.org/10.1109/ICDCS.2012.17

[Zhao and Strom 2001] Yuanyuan Zhao and Rob Strom. 2001. Exploitng Event Stream Interpretation in Publish-subscribe Systems. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing (PODC '01)*. ACM, New York, NY, USA, 219–228. DOI:http://dx.doi.org/10.1145/383962.384023