Event Correlation: Language and Semantics

César Sánchez, Sriram Sankaranarayanan, Henny Sipma,
Ting Zhang, David Dill, and Zohar Manna

Computer Science Department
Stanford University
{cesar,srirams,sipma,tingz,dill,zm}@CS.Stanford.EDU

Abstract. Event correlation is a service provided by middleware platforms that allows components in a publish/subscribe architecture to subscribe to patterns of events rather than individual events. Event correlation improves the scalability and performance of distributed systems, increases their analyzability, while reducing their complexity by moving functionality to the middleware. To ensure that event correlation is provided as a standard, reliable service, it must come with a well-defined, unambiguous semantics.
In this paper we present a language and formal model for event correlation with an operational semantics defined in terms of transducers. The language has been motivated by an avionics application and includes constructs for modes in addition to the more common constructs such as selection, accumulation and sequential composition. We show how the transducers can be constructed in a compositional way from event correlation expressions. Prototype event processing engines for this language have been implemented in both C++ and Java and are being integrated with third-party event channels.

1 Introduction

Publish-subscribe is an event-based model of distributed systems that decouples event publishers from event consumers. Event consumers register their interests with the middleware by means of a subscription policy. Events published to the middleware are delivered only to those consumers that expressed an interest. The publish-subscribe paradigm is useful in practice for building large systems in which not all components are known at design time, or components may be dynamically added at runtime such as in mobile systems. These systems may even extend world wide. In [5] a world is predicted in which pervasive computing devices generate 10,000,000,000 events per second, with billions of subscribers from all over the planet. Clearly, extensive filtering and correlation are required at multiple levels and locations to manage these volumes.

Several middleware platforms exist that provide event notification services. Examples include Gryphon [1], Ace-Tao [20], Siena [3], Elvin [2]. An overview and comparison of such systems is given in [16]. In some middleware platforms, such as the real-time event channel (RTEC) of ACE-TAO [20], consumers subscribe with the middleware with a list of event types or sources they wish to receive. Suppliers publish their events to the RTEC, which then distributes them to the consumers that signed up for them. This service, also called event filtering, reduces unnecessary component activation and simplifies construction and configuration of complex systems.

Event correlation extends event filtering offering consumers the capability to register with the RTEC with more complex subscriptions, including combinations of events and temporal patterns. Providing event correlation as a standard middleware service further enhances the performance of embedded applications. However, more importantly, it enables the transfer of functionality from application components to the middleware, which

- reduces software development and maintenance cost by decreasing the number of special-purpose components to be developed, as complexity is factored out of the component into the middleware;
- increases reliability, as this service can be verified and tested once as part of the platform and reused many times;
- increases the analyzability of the system: where event dependencies were before largely hidden inside application components, with event correlation provided, these dependencies are available explicitly to analysis tools in the form of event correlation expressions with well-defined semantics;
- increases the accuracy of schedulability analysis by taking into account dependencies between component invocations, enabling more efficient utilization of computing and network resources;
- increases flexibility in configuration: event correlation expressions can be changed more easily at runtime or on short notice than components can be replaced.

To enable event correlation to be provided as a standard, reliable middleware service, it must come with a well-defined, unambiguous semantics. In addition, it would be desirable if event-processing code could be generated automatically from the (declarative) event correlation expressions, thus ensuring adherence to the semantics by construction.

In this paper we present a language and a computational model for event correlation that provides an unambiguous semantics for event correlation expressions. The model is based on automata/transducers, a well-studied domain with a large body of analysis methods and tools. Another attractive property of transducers is that it is an operational model: they can be used directly in the RTEC to process events interpretatively, or used to automatically generate the code to process the events.

The development of this language was initiated to enable the use of event correlation in the Boeing Open Experimental Platform, a component of Boeing's Bold Stroke avionics middleware architecture [21]. Some of the constructs included in the language were directly motivated by the Boeing product scenarios.

The paper is organized as follows. In section 2 we give an intuitive overview of the features of a simple version of our event correlation expression language. In section 3 we present the new model of *correlation machines* and its extension *correlation modules* that are used to define the operational semantics of the constructs in the correlation language. This translation is described in section 4. In section 5 we briefly present some related work. Finally, section 6 contains the conclusions and outlines some directions for future research.

2 Event Correlation Language

Syntax An ECL expression is constructed out of predicate formulas, to which we apply the combination operators shown below.

A predicate formula is itself a correlation expression. If $\phi, \phi_1 \ldots \phi_n$ are correlation expressions then so are

$\mathbf{accumulate}\{\phi_1, \ldots, \phi_n\}$	$\mathbf{fail}\{\phi\}$
$\mathbf{select}\{\phi_1, \ldots, \phi_n\}$	$\mathbf{push}(x)\{\phi\}$
$\mathbf{sequential}\{\phi_1, \ldots, \phi_n\}$	$\mathbf{persist}\{\phi\}$
$\mathbf{do}\{\phi_1\}\mathbf{unless}\{\phi_2\}$	$\mathbf{repeat}\{\phi\}$
$\mathbf{parallel}\{\phi_1, \ldots, \phi_n\}$	$\mathbf{loop}\{\phi\}$

An informal description of each of the constructs follows.

In general, an expression is evaluated over an input stream of events. On any event the evaluation may complete successfully, it may complete in failure, or may not complete.

Basic Constructs : The lowest-level construct is a *simple event predicate* on a single event. This event predicate may range from being an enumeration on a finite alphabet to first-order predicates on the source, timestamp, type, and data content of the event. In the context of the language described here we are only concerned with whether or not an event satisfies the predicate. In general we assume we have an effective way of deciding this. To simplify the presentation in this paper we here assume a finite alphabet of input events.

The evaluation of a predicate expression completes successfully when the event received satisfies the predicate. It does not complete on any other event. In particular, the expression never fails: events that do not satisfy the predicate are simply ignored.

Simple predicate expressions may be combined using the standard boolean operators with the usual semantics. Thus these compound predicate expressions are also evaluated over a single event. An example of a compound predicate expression is $\langle source = GPS \rangle \wedge \langle type = DATA \rangle$ which is satisfied by any event that meets both conditions.

The boolean constants **true** and **false** are available as trivial predicate expressions. The predicate **true** completes successfully upon the reception of any event; **false** is never satisfied, and hence never completes. Note that **false** does not complete in failure either (it simply blocks waiting for an event satisfying "false").

Simple correlator expressions : Predicate expressions can be combined into *correlator expressions* that may need to consume multiple events before they complete.

The most commonly seen event correlation constructs are the accumulation and selection operators, with various semantics. In our semantics the evaluation of an accumulation expression evaluates all subexpressions in parallel. It completes successfully when all subexpressions have completed successfully, and it completes with failure when one of the subexpressions results in failure. The selection operator is the dual of the accumulation operator. Here the subexpressions are also evaluated in parallel, but the evaluation completes if one of the subexpressions completes successfully and fails if all subexpressions complete in failure. An alternative semantics of selection, denoted by **select*** is sometimes useful, in which the evaluation completes in failure when one of the subexpressions ends in failure.

A sequential operator is less commonly seen in the popular middle-ware platforms. However it is useful in forming more complex patterns. The evaluation of a sequential expression proceeds sequentially, where the evaluation of ϕ_{i+1} is started after successful completion of ϕ_i. The expression completes successfully upon successful completion of ϕ_n. It ends in failure when any of the subexpressions ends in failure.

None of the constructs presented so far can lead to completion in failure. (Note that the accumulation and selection operator can propagate failure, but not cause it.) The do-unless operator is the first operator that can cause an expression to end in failure. It was inspired by the, less general, sequential unless operator in GEM [15]. The evaluation of the do-unless expression evaluates the two subexpressions in parallel. It completes successfully when ϕ_1 completes successfully, it completes with failure when either ϕ_1 completes in failure or ϕ_2 completes successfully. Note that completion of ϕ_2 in failure does not affect the evaluation of ϕ_1. The do-unless expression is useful to preempt other expressions, especially those that do not complete by themselves, some of which are described below.

The *fail* operator has been included in the language to allow the invertion of its argument. That is, a fail expression completes successfully when ϕ completes in failure and vice versa.

Repetition : The language provides several constructs for repetition, which may be parameterized by the maximum of number of repetitions. The constructs differ in their handling of failure of the subexpressions. The evaluation of a repeat expression repeats ϕ, irrespective of its failure or success. With a persist expression, ϕ is repeated until success, while a loop expression repeats ϕ until failure.

Generating output : The expressions presented so far do not generate any output. The purpose of the language is to provide a means to notify the consumer that certain patterns of events have occurred, and therefore specific operators are introduced to generate output. The simplified version of the language presented in this paper does not support the forwarding of events. The only output that can be generated are tokens from an alphabet of constants. The push expression $\textbf{push}(x)\{\phi\}$ outputs character x upon the successful completion of ϕ, after which it completes successfully. If ϕ completes with failure no output is generated and the push expression itself completes with failure.

Parallel Expressions : Correlator expressions may be combined into parallel expressions. In the previous subsection, several subexpressions were said to be evaluated in parallel. However, their evaluations were linked in the sense that completion of one of them affected the completion or termination of the others.

On the other hand, in a parallel expression, the subexpressions are evaluated in parallel and independent of each other. Thus, the completion of any subexpression does not affect the evaluation of other subexpressions. A parallel expression never completes, but it can be preempted by, for example, an unless expression. Typically, a parallel expression would contain multiple **repeat** subexpressions.

Mode Expressions : Mode expressions were directly motivated by the avionics applications that initiated this work. Mode expressions provide a convenient way to support different modes of operation. They allow simultaneous mode switching of multiple components in a system without any direct coordination between these components.

A mode expression has multiple modes, each consisting of a predicate expression called the *mode guard*, and a correlator expression. The mode guards are expected to be mutually exclusive, such that at any time exactly one mode is active.

$$\begin{array}{l}\textbf{in}\ (p_1)\ \textbf{do}\ \{\phi_1\}\\ \textbf{in}\ (p_2)\ \textbf{do}\ \{\phi_2\}\\ \dots\\ \textbf{in}\ (p_n)\ \textbf{do}\ \{\phi_n\}\end{array}$$

A mode i is activated when its mode-guard p_i completes successfully. Upon activation of a new mode, evaluation of the expression in the current mode is terminated, and the evaluation of the expression associated with the new mode is started. Like the parallel expression, the mode expression does not complete by itself, but it can be preempted by other expressions.

3 Correlation Machines

ECL expressions can be viewed as temporal filters: for a given input sequence of events they specify what must be transmitted to the consumer, and when.

Definition 1 (Input-Output pair). *Let Σ_{in} be a finite alphabet of input events and Σ_{out} be a finite alphabet of output constants. Let $\sigma : e_1, e_2, \dots$ be a sequence of events with $e_i \in \Sigma_{in}$, and $\kappa : o_1, o_2, \dots$ be a sequence of output constants with $o_i \in \Sigma_{in}$. An* input-output pair *is a pair $(\sigma, \langle\kappa, f\rangle)$ with $f : N^+ \mapsto N$ a weakly increasing function that specifies the position of the outputs relative to* *the input sequence. That is, for each $i > 0$, o_i is produced while or after the event $e_{f(i)}$ is consumed, and strictly before $e_{f(i)+1}$ (if present) is consumed.*

Example 1. The input-output pair $(\sigma, \langle\kappa, f\rangle)$ with

$$\begin{array}{ll} \sigma : e_1, e_2, e_3, e_4, e_5 & \\ \kappa : o_1, o_2 & \end{array} \qquad f(1) = 2, \quad f(2) = 4$$

specifies that output o_1 should be produced between the consumption of events e_2 and e_3, and output o_2 should be produced after consumption of event e_4.

Although it may be interesting to define the semantics of ECL expressions directly in terms of input-output pairs, we have found it more practical to define the semantics operationally in terms of transducers. The main advantage of this approach is that it immediately provides a standard implementation for each correlator expression.

To facilitate a compositional definition of the semantics of ECL expressions we introduce the *correlation machine*, a finite-state transducer extended with facilities for concurrency and reset in a way similar to Petri Nets [19], that support a concise representation of simultaneous evaluation and preemption. Concurrency is provided by having transitions that map sets of sets of states into sets of states, such that multiple states may have to be active for the transition to be enabled. Reset is provided by explicitly including a *clear set* in the transition, which may be a superset of the enabling condition. A similar way of reset was also proposed in [22].

The addition of concurrency makes the transducer potentially nondeterministic, which is undesirable for an operational model. In the compositional construction of the correlators we have found it convenient to eliminate this nondeterminism by means of a partial order on transitions that specifies which of the enabled transitions have priority.

Correlation machines may have internal transitions, that is, transitions that do not consume any input events. To enable uniform treatment of all transitions we define expanded input and output sequences that are padded with empty input events and empty output constants in such a way that the constraints on the relative positions of input and output are preserved.

We will use $\overline{\Sigma_{in}}$ as a short for $\Sigma_{in} \cup \{\epsilon\}$, and $\overline{\Sigma_{out}}$ as a short for $\Sigma_{out} \cup \{\epsilon\}$.

Definition 2 (Expanded input-output pair). *The pair (σ', κ') is an expansion of $(\sigma, \langle\kappa, f\rangle)$ if σ' is equal to σ interleaved with finite sequences of the empty input event ϵ, and κ' is equal to κ interleaved with finite sequences of the empty output event ϵ. Let g, h be the (weakly increasing) functions that map the indices of the elements in σ, κ into the indices of the corresponding elements in σ', κ'. We say that the expansion respects f if the outputs in the expansion are produced "at the right time", that is, if for all $i > 0$*

$$g(f(i)) \leq h(i) < g(f(i) + 1)$$

Example 2. For the sequences σ, κ in example 1 the expansion

$$\begin{array}{l} \sigma' : e_1\ \epsilon\ \epsilon\ e_2\ \ \epsilon\ \ \epsilon\ e_3\ e_4\ e_5\ \epsilon\ \epsilon \\ \kappa' :\ \epsilon\ \ \epsilon\ \epsilon\ \ \epsilon\ \ o_1\ \epsilon\ \ \epsilon\ \ o_2\ \ \epsilon\ \ \epsilon\ \epsilon \end{array}$$

respects f, as $g(f(1)) = g(2) = 4$, $g(f(1)+1) = g(3) = 7$, and $h(1) = 5$ for the first output, and $g(f(2)) = g(4) = 8$, $g(f(2)+1) = g(5) = 9$, and $h(2) = 8$ for the second output.

Definition 3 (Correlation Machine). *A* correlation machine $\Psi : \langle Q, I, \mathcal{T}, \prec\rangle$ *has the following components*

- *Q: a finite set of states*
- *$I \subseteq Q$: the set of initial states,*
- *$\mathcal{T}$: a finite set of transitions $\tau = (En, a, Clr, Tgt, o) \in \mathcal{T}$, with $En \subseteq 2^Q$, the enabling states, $Clr \subseteq Q$, the clear states, $Tgt \subseteq Q$, the target states, and $a \in \overline{\Sigma_{in}}$, the input event, and $o \in \overline{\Sigma_{out}}$, the output character, and*
- *$\prec \subseteq \mathcal{T} \times \mathcal{T}$: a partial order on transitions.*

Definition 4 (Behavior). *An input-output pair $(\sigma, \langle\kappa, f\rangle)$ is a* behavior *of a correlation machine $\Psi : \langle Q, I, \mathcal{T}, \prec\rangle$ if there exists an expansion (σ', κ'),*

$$\begin{array}{l} \sigma' : y_0\ \ y_1\ \ y_2\ \ y_3\ \ldots \\ \kappa' : w_0\ w_1\ w_2\ w_3\ \ldots \end{array}$$

that respects f, and if there exists a sequence of sets of states and sets of transitions

$$S_0, T_0, S_1, T_1, \ldots$$

such that

Initiation *(I): $S_0 = I$*
Consecution *: for each $j \geq 0$*

(C0) all transitions in T_j are enabled, that is, for all $\tau = (En_\tau, \ldots) \in T_j$, for all sets $s \in En_\tau$ at least one state q is in the current set of states:

$$\forall s \in En_\tau \quad \exists q \in s\ .\ q \in S_j$$

(C1) all transitions $\tau \in T_j$ are taken:

$$S_{j+1} = (S_j - \bigcup_{\tau \in T_j} Clr_\tau)\ \cup \bigcup_{\tau \in T_j} Tgt_\tau$$

(C2) all transitions $\tau \in T_j$ are minimal in the partial order with respect to all transitions that are enabled on S_j, that is, for all $\tau' \in \mathcal{T}$, $\tau' \prec \tau$

$$\exists s \in En_{\tau'} \quad \forall q \in s\ .\ q \notin S_j$$

and T_j is maximal, that is, it contains all *transitions that are enabled and minimal with respect to the partial order.*

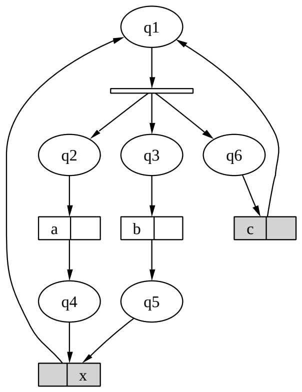


Fig. 1. Correlation automaton

(C3) for all transitions $\tau = (\ldots, a, \ldots, o) \in T_j$, the input event a agrees with the input event in the input sequence in σ', and the output o agrees with the output constant in κ', that is, $a = y_j$ and $o = w_j$.

Several transitions can be run in parallel. Condition *C0* says that all of them have to be enabled, while condition *C3* establishes that they have to be consistent with the input and output. Conditions *C1* and *C2* establish that the set of fired transitions is the greatest set of enabled transitions that are minimal with respect the partial order.

Note, in particular, that if in state S_j no transition is enabled, or no enabling transition is interested in input event y_j then $T_j = \emptyset$ and $S_{j+1} = S_j$.

Example 3. Consider the correlation machine $\mathcal{A} = \langle Q, I, \mathcal{T}, \prec \rangle$ with components

– $Q = \{q_1, q_2, q_3, q_4, q_5, q_6\}$, with $I = \{q_1\}$,
– $\mathcal{T} = \{\tau_1, \tau_2, \tau_3, \tau_4, \tau_5\}$ with

$$\begin{aligned}
\tau_1 &= (\{q_1\}, \epsilon, \{q_1\}, \{q_2, q_3, q_6\}, \epsilon) \\
\tau_2 &= (\{q_2\}, a, \{q_2\}, \{q_4\}, \epsilon) \\
\tau_3 &= (\{q_3\}, b, \{q_3\}, \{q_5\}, \epsilon) \\
\tau_4 &= (\{\{q_4\}, \{q_5\}\}, \epsilon, Q, \{q_1\}, x) \\
\tau_5 &= (\{q_6\}, c, Q, \{q_1\}, \epsilon)
\end{aligned}$$

– $\prec = \emptyset$.

Remark: When the enabling condition of a transition consists of a single set we write just the set rather than the set of that set, to avoid clutter in notation.

A graphical representation of the machine is shown in figure 1. In the figure transitions are shown by rectangles. The automaton produces an output x after every a and b in any order, without an intervening c. For example, the event sequence *aabbcbac* produces the run and output sequence shown in figure 2. Notice the reset effect of transitions τ_4 and τ_5. Both clear all currently active states and start afresh.

σ :				a		a		b						b		c				b		a						c			
π :	q_1	τ_1	q_2 q_3 q_6	τ_2	q_4 q_3 q_6	$\emptyset$	q_4 q_3 q_6	τ_3	q_4 q_5 q_6	τ_4	q_1	τ_1	q_2 q_3 q_6	τ_3	q_2 q_5 q_6	τ_5	q_1	τ_1	q_2 q_3 q_6	τ_3	q_2 q_5 q_6	τ_2	q_4 q_5 q_6	τ_4	q_1	τ_1	q_2 q_3 q_6	τ_5	q_1	τ_1	q_2 q_3 q_6
$\mathcal{O}$:										x														x							

Fig. 2. Run of $\mathcal{A}$ on event sequence *aabbcbac*

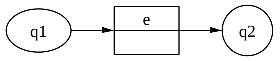


Fig. 3. Correlation module for a single event e

Correlation Module Correlation machines for correlation expressions are constructed in a bottom-up fashion from the subexpressions. The building block, called a *correlation module* is a correlation machine extended with two final states: s to indicate successful completion and f to indicate failure of a subexpression. Final states only have significance in the modular construction; they become regular states in the final machine.

4 Translation

In this section we introduce the correlation module for some of the constructs defined in section 2.

Single input event : The correlation module for a single input event $e \in \Sigma_{in}$ is shown in figure 3. It has three states, $\{q_1, q_2, q_3\}$, of which q_1 is initial, and one transition,

$$\langle\{q_1\}, e, \{q_1\}, \{q_2\}, \epsilon\rangle$$

that takes the given event as input and moves to the success state without producing any output. The success state is q_2 and the failure state is q_3. Notice that the failure state is not reachable, reflecting the fact that a predicate expression cannot complete in failure.

Compound Expressions : The description of the construction of correlation modules for compound expressions assumes the correlation modules $\mathcal{M}_1, \ldots \mathcal{M}_n$ for the subexpressions have already been constructed. We describe the construction for the accumulation expression in some detail to illustrate the construction method and mostly rely on the figures for the other constructs.

The correlation module $\mathcal{M} = \langle Q, I, \mathcal{T}, \prec, s, f\rangle$ for an accumulation expression **accumulate**$\{\phi_1, \phi_2\}$ with correlation modules $\mathcal{M}_1$ and $\mathcal{M}_2$ for ϕ_1 and ϕ_2 respectively consists of the following components:

– The set of states is the union of the set of states for the subexpressions extended with two new states (not appearing in Q_1 or Q_2) to be the new success and failure states. The set of initial states is set to reflect that both subexpressions should be evaluated in parallel:

$$Q \;=\; Q_1 \cup Q_2 \cup \{q_1, q_2\} \quad \text{with} \quad I \;=\; I_1 \cup I_2$$

– The set of transitions is the union of the sets of transitions of the subexpressions, extended with two new internal transitions that complete the evaluation of the accumulation expression:

$$\mathcal{T} \;=\; \mathcal{T}_1 \cup \mathcal{T}_2 \cup \{\tau_1, \tau_2\}$$

with

$$\begin{array}{l}\tau_1 = (\{\{s_1\},\{s_2\}\}, \epsilon, Q_1 \cup Q_2, \{q_1\}, \epsilon)\\ \tau_2 = (\{f_1, f_2\}, \epsilon, Q_1 \cup Q_2, \{q_2\}, \epsilon)\end{array}$$

Notice that the enabling condition of τ_1 is conjunctive: both states must be present, while the enabling condition of τ_2 is disjunctive: the transition is enabled if one of the states is present. Both transitions terminate evaluation of the entire expression by clearing all states in both submodules.

– The partial orders of the two subexpressions are combined and the new transitions are given lower priority

$$\prec \;=\; \prec_1 \cup \prec_2 \cup (\mathcal{T}_1 \cup \mathcal{T}_2, \{\tau_1, \tau_2, \tau_3\}) \;\cup \{(\tau_1, \tau_2)\}$$

to reflect that internal transitions of the subexpressions must always be taken before the internal transitions of this module, to make sure the subexpressions have finished their "cleaning up". The pair (τ_1, τ_2) is added to eliminate the potential nondeterminism if these transitions become enabled simultaneously: it gives priority to success.

– The final states are the two newly added states $s = q_1$ and $f = q_2$.

The resulting correlation module is illustrated in figure 4(a).

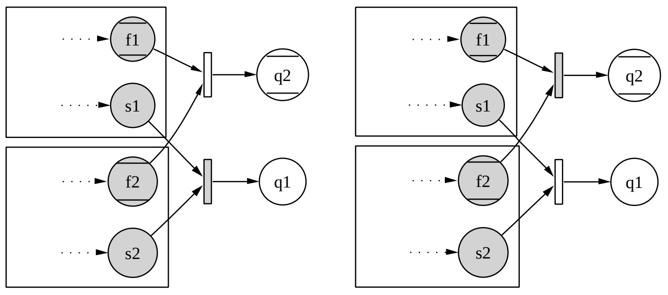
The correlation module for the selection expression **select**$\{\phi_1, \phi_2\}$ is very similar to that of the accumulation expression. As mentioned before, the selection expression is the dual of the accumulation expression, which is reflected in the dualization of the transitions τ_1 and τ_2:

$$\begin{array}{l}\tau_1 = (\{s_1, s_2\}, \epsilon, Q_1 \cup Q_2, \{q_1\}, \epsilon)\\ \tau_2 = (\{\{f_1\},\{f_2\}\}, \epsilon, Q_1 \cup Q_2, \{q_2\}, \epsilon).\end{array}$$

The resulting correlation module is illustrated in figure 4(b).

The correlation module for sequential composition **sequential**$\{\phi_1, \phi_2\}$ is shown in figure 5. It adds two new states, the success state q_1 and the failure state q_2, and four internal transitions

$$\begin{array}{l}\tau_1 = (\{s_1\}, \epsilon, Q_1, I_2, \epsilon)\\ \tau_2 = (\{f_1\}, \epsilon, Q_1, \{q_2\}, \epsilon)\\ \tau_3 = (\{s_2\}, \epsilon, Q_2, \{q_1\}, \epsilon)\\ \tau_4 = (\{f_2\}, \epsilon, Q_2, \{q_2\}, \epsilon)\end{array}$$



(a) accumulation (b) selection

Fig. 4. Correlation modules for accumulation and selection. The shaded circles denote the success and failure nodes of the modules $\mathcal{M}_{\{1,2\}}$, and the unshaded circles denote the success and failure nodes of the resulting module. Transitions with conjunctive enabling condition are shown shaded and disjunctive transitions are unshaded.

where τ_1 links the success state of the first module to the initial states of the second module.

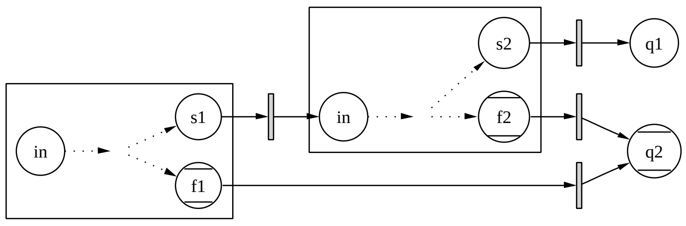


Fig. 5. Correlation module for sequential composition

The correlation module for an unless expression, $\mathbf{do}\{\phi_1\}$ $\mathbf{unless}\{\phi_2\}$ is shown in figure 6(a). The two new transitions have transition relation

$$\tau_1 = (\{s_1\}, \epsilon, Q_1 \cup Q_2, \{q_1\}, \epsilon)$$
$$\tau_2 = (\{f_1, s_2\}, \epsilon, Q_1 \cup Q_2, \{q_2\}, \epsilon)$$

where transition τ_1 is the success transition, while τ_2 leads to failure. As mentioned in section 2, the unless expression can cause an expression to fail, as witnessed by τ_2, which leads from s_2, a success state, to q_2, a failure state.

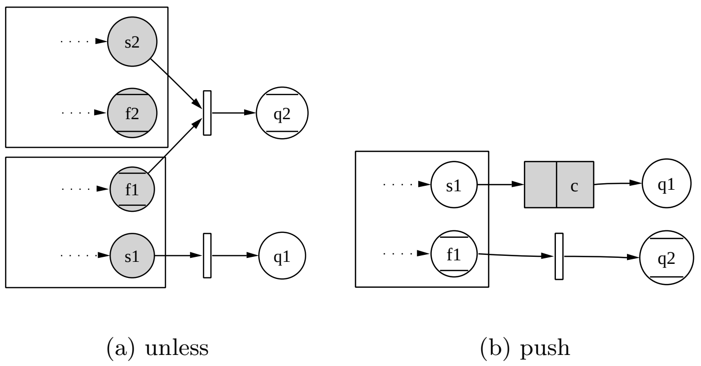


Fig. 6. Correlation modules for unless and push.

Output Expressions : The correlation module for the push expression **push** (x) $\{\phi\}$, shown in figure 6(b), adds two transitions

$$\tau_1 = (\{s_1\}, \epsilon, Q_1, \{q_1\}, x)$$
$$\tau_2 = (\{f_1\}, \epsilon, Q_1, \{q_2\}, \epsilon)$$

the first of which outputs constant x upon successful completion of the module for ϕ, while τ_2 simply propagates the failure.

Mode Expressions : The correlation module $\mathcal{M} = \langle Q, I, \mathcal{T}, \prec, s, f\rangle$ for a mode expression with correlation modules $\mathcal{M}_i = \langle Q_i, I_i, \mathcal{T}_i, \prec_i, s_i, f_i\rangle$ for ϕ_i, $i = 1, \ldots n$, and $\mathcal{M}_{gi} = \langle Q_{gi}, I_{gi}, \mathcal{T}_{gi}, \prec_{gi}, s_{gi}, f_{gi}\rangle$ for p_i, $i = 1 \ldots n$ consists of the following components:

- The set of states is the union of the states of the guards and the expressions and two additional states for the new success and failure state. The initial states are those of the first expression, ϕ_1, combined with the initial states of the guards of the other expressions

$$Q = \bigcup_{i=1}^{n} Q_i \cup \bigcup_{i=1}^{n} Q_{gi} \cup \{q_1, q_2\} \quad \text{with } I = I_1 \cup \bigcup_{i=2}^{n} I_{gi},$$

- The set of transitions includes the transitions of the expressions and the guards, and is extended with one *mode entry transition* τ_i for each mode ϕ_i, $i = 1 \ldots n$ with transition relation

$$\tau_i = (\{s_{gi}\}, \epsilon, Q, I_i \cup G_i, \epsilon) \quad \text{with } G_i = \bigcup_{1 \leq j \leq n, j \neq i} I_{gj}$$

  where G_i is the union of the set of initial states of the guards other than p_i. Similar to the module for the parallel expression, both the success and failure state of this module are unreachable.

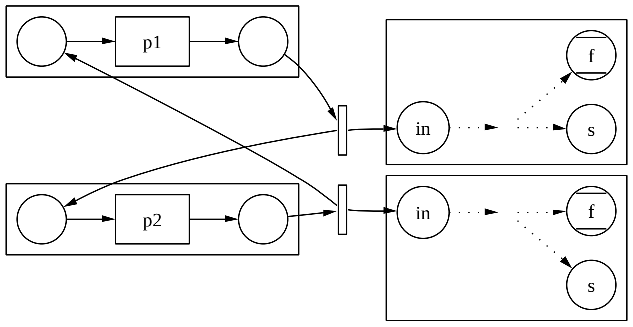


Fig. 7. Correlation module for mode expression

- The partial orders of all modules are combined, and the transitions belonging to the guard expressions are given higher priority than those belonging to the expressions, to reflect that the preemption of a mode by another mode has higher priority than the consumption of the same event within a mode.
- The final states are $s = q_1$ and $f = q_2$.

5 Related Work

Several proposals for specification languages for event correlation appear in the literature. In [4], a language based on typed λ-calculus is used. Therein, composite events are represented by *curried functional expressions*. Event arrivals correspond to expression evaluation, while completed *normal-form* expressions are delivered to consumers. A formal semantics is given in terms of reduction rules.

Zhu and Sethi [24] propose an event correlation language that includes a negation operator. Expressions are evaluated relative to some fixed or sliding time window; the negation operator precludes the occurrence of its argument during the time window. Hinze and Voisard [11] use a *parameterized event algebra* to describe the semantics of composite events. Negation is interpreted as in [24]. Parameters are used to specify the handling of multiple instances of events that belong to the same class, with the options to choose all the events, the first event, or the last event, in the collection. Parameters also specify whether events may be used multiple times, or consumed on matching.

Zhang and Unger [23] present a composite event specification language with operators for disjunction, conjunction, sequence and counter-based events within a moving window. The semantics is given declaratively, but the result of an expression is not guaranteed to be unique. Hence, *decorators* are added to choose from the subsets of events that satisfy an expression. COBEA [14] is an event-based architecture that includes an evaluation engine for composite events specified in the Cambridge Composite Event Language [9, 10]. The semantics is defined in terms push-down machines.

The use of composite temporal events has received much attention in the active database community. Gehani et al. [7] propose a language for specifying composite events with semantics in terms of event history maps. This language is expressively equivalent to regular expressions, that can be translated into NFA for event notification. Coordination of subexpressions is done through correlation variables to allow parameterization. The method is implemented in COMPOSE [6]. In [17, 18] Motakis and Zaniolo propose a specification language based on Datalog. Their pattern language allows parameterization, with parameter instantiations propagated through the expression.

6 Conclusion and future directions for research

We have presented a declarative language to express event correlation expressions for publish-subscribe systems. The semantics of this language was defined in terms of correlation machines, an operational model that can be directly used as an event processing engine in an event channel.

Applications and Implementation : The event correlation language presented here has been applied in Boeing's Open Experimental Platform (OEP). It was found that the use of event correlation expressions reduced the need for special-purpose components by moving functionality to the event channel. A prototype event processor based on correlation machines has been implemented in C++ and is currently being integrated in the OEP. A separate event processor has been implemented in Java and has been integrated with FACET [13], a real-time event channel being developed at Washington University at St Louis.

Analysis : Publish-subscribe systems are used in mission-critical avionic applications [20] and may potentially be used in many other safety-critical systems. The availability of analysis tools for event correlation expressions will greatly contribute to the acceptance of this technology as a reliable addition to simple event filtering. Some of the questions that may be asked about the behavior of these expressions include:

– checking expressions for *triviality*, (i.e. whether the expression filters in everything), or *vacuity* (where the expression rejects everything),
– checking *liveness*, that is, checking that at any point in the evaluation of an expression, it should be possible for some event-sequence to lead the machine to acceptance,
– checking containment among correlators or systems of correlating expressions,
– checking correlation expressions against event-loops for equivalence,
– checking event dependencies.

Since these machines are essentially finite state, we believe that many of these problems are tractable.

Optimization : The correlation machines that are generated by the construction method described in Section 4 are obviously rather inefficient; they contain unreachable states and redundant internal transitions that can be eliminated. Other transformations may be envisaged for time/space trade-offs. Our current model favors a concise representation of the correlation machine. However this comes at the price of increased event processing time. In time-critical applications one may want to eliminate the concurrency and fully determinize the transducer, thus reducing event-processing time, but increasing the space required to store the machine. Several intermediate solutions may exist.

Evaluation Strategies : A large publish subscribe system may have many components with numerous subscriptions. Hence, the middleware is faced with the task of evaluating each of these expressions for each of the events. This can cause severe overhead on the middleware, degrading its performance. There are two complementary approaches to alleviate this:

The first tactic is that of that of *composing* correlation machines. If a number of consumers subscribe with different correlation policies, a naive implementation would run all the correlators in parallel. A more efficient version should try to *reuse* the work of the evaluation of different machines. In [1] a first approach to this problem has been considered. This problem resembles that of performing multiple searches in parallel in a string [8].

The second tactic is to *decompose* a machine into many machines and distribute these along the network. Thus, if at some point in the routing of an event, a determination can be made that the event is not of interest for a set of consumers, then the routing of the event can be restricted. This can save network bandwidth and yield more processing time. This problem has been called the *quenching problem* in the related literature [12].

References

1. M. Aguilera, R. Storm, D. Sturman, M. Astley, and T. Chandra. Matching events in a content based subscription system. In *PODC*, 1999.
2. B.Segall and S. Arnold. Elvin has left the building: A publish/subscribe notification service with quenching. In *Queensland AUUG Summer Technical Conference, Brisbane, Australia*, 1997.
3. A. Carzaniga, D. S. Rosenblum, and A. L Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, August 2001.
4. S. Courtenage. Specifying and detecting composite events in content-based publish/subscribe systems. In *Proc. DEBS'02*, 2002.
5. J. Crowcroft, J. Bacon, P. Pietzuch, G. Coulouris, and H. Naguib. Channel islands in a reflective ocean: Large-scale event distribution in heterogeneous networks. *IEEE Communications Magazine*, (9):112–115, September 2002.
6. N.H. Gehani, H.V. Jagadish, and Oded Shmueli. COMPOSE. a system for composite event specification and detection. Technical report, AT&T Bell Laboratories, 1992.

7. N.H. Gehani, H.V. Jagadish, and Oded Shmueli. Composite event specification in active databases: Model and implementation. In *Proceeding VLDB'92*, pages 327–338, 1992.
8. D. Gusfield. *Algorithms on strings, trees and sequences.* Cambridge Univ. Press, 1997.
9. R. Hayton, J. Bacon, J. Bates, and K. Moody. Using events to build large scale distributed applications. In *Proc. ACM SIGOPS European Workshop*, pages 9–16. ACM, September 1996.
10. Richard Hayton. *OASIS. An Open Architecture for Secure Internetworking Services.* PhD thesis, Fitzwilliam College, University of Cambridge, 1996.
11. A. Hinze and A. Voisard. A flexible parameter-dependent algebra for event notification services. Technical Report TR-B-02-10, Freie Universität Berlin, 2002.
12. Y. Huang and H. Garcia-Molina. Publish/subscribe in a mobile environment. In *Proc.* MobiDE, 2001.
13. Frank Hunleth, Ron Cytron, and Christopher Gill. Building customizable middleware using aspect oriented programming. In *Workshop on Advanced Separation of Concerns (OOPSLA'01)*, 2001.
14. Chaoying Ma and Jean Bacon. COBEA: A CORBA-based event architecture. In *Proc. USENIX COOTS'98*, pages 117–131, April 1998.
15. M. Mansouri-Samani and M. Sloman. GEM, a generalised event monitoring language for distributed systems. In *Proceedings of ICODP/ICDP'97*, 1995.
16. R. Meier. State of the art review of distributed event models. Technical report, University of Dublin, Trinity College, 2000.
17. I. Motakis and C. Zaniolo. Composite temporal events in active database rules: A logic-oriented approach. In *Proceedings of DOOD'95*, volume 1013 of *LNCS*, pages 19–37. Springer-Verlag, 1995.
18. I. Motakis and C. Zaniolo. Formal semantics for composite temporal events in active database rules. *Journal of Systems Integration*, 7(3–4):291–325, 1997.
19. W. Reisig. *Petri Nets: An Introduction*, volume 4 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, 1985.
20. D. Schmidt, D. Levine, and T. Harrison. The design and performance of a real-time CORBA object event service. In *Proceedings of OOPSLA '97*, 1997.
21. David Sharp. Reducing avionics software cost through component based product line development. In *Proceedings of the Software Technology Conference*, 1998.
22. T. Vesper and M. Weber. Structuring with distributed algorithms. In *Proceeding of CS&P98*, September 1998.
23. R. J. Zhang and E. Unger. Event specification and detection. Technical Report TR CS-96-8, Kansas State University, June 1996.
24. D. Zhu and A. S. Sethi. SEL, a new event pattern specification language for event correlation. In *Proc. ICCCN-2001,*, pages 586–589, October 2001.