# SEL, A New Event Pattern Specification Language for Event Correlation

Dong Zhu, Adarshpal S. Sethi
University of Delaware
Department of Computer and Information Sciences
Newark, Delaware 19716
{*dzhu,sethi*}*@cis.udel.edu*

*Abstract*— **Event pattern detection is the one of the major techniques used for event correlation in network and distributed systems management. This paper focuses on the design issues of event pattern specification languages. The discussion is organized around the event operators in existing event languages that we think are problematic and around the temporal specification aspect. Semantic issues are discussed and various languages are investigated. The study has revealed weaknesses of design in semantic appropriateness and completeness of certain event operators, in flexibility of operator usage and timing specification, in effectiveness and efficiency of expressions, and in readability of the languages. Based on the findings, we propose a new event language called SEL, which attempts to avoid some of these problems. SEL is novel in its negation operator usage, the way followed-by semantics is provided, and how composite event time is determined in the presence of the negation operator. It is comprehensive yet relatively simple and intuitive to use. Expressions written in SEL appear to be very readable and easy to maintain.**

*Keywords*— **event correlation, network management, event pattern detection, composite event, event expression**

## I. Introduction

Event monitoring and analysis is crucial for managing networks and distributed systems. As one of the primary techniques adopted for this purpose, event correlation [1], [2], [3] aims at deriving more concise and meaningful information from potentially large volumes of lower level events. Detecting event patterns expressed in the form of event expressions is a common event correlation technique. This paper proposes a novel event pattern specification language, SEL.

Event patterns are specified as event expressions which consist of events connected by event operators, constraints on event attributes expressed as predicates on the attributes, and constraints on time windows which delimit occurrence time of the patterns. Event languages are inherently more complex than *regular expressions* because events are dynamic, events have attributes, and event languages need to provide means so that sophisticated timing relationship between events can be specified.

Our study of existing event pattern languages has revealed weaknesses of design in semantic appropriateness and completeness of certain common event operators, in flexibility of operator usage and timing specification, in effectiveness and efficiency of expressions, and in readability of the languages. Based on these findings, we propose a new event language, SEL, which tries to avoid some of these problems. SEL is novel in its negation operator usage, the way followed-by semantics is provided, and how composite event time is determined in the presence of the negation operator. It is comprehensive yet simple and intuitive to use. Expressions written in SEL appear to be very readable and easy to maintain. The language has shown its usefulness in SHAMAN network management scripting framework [4] and SDB, a SHAMAN application debugger [5]. The rest of the paper discusses the motivation behind SEL and briefly describes its syntax and semantics. More details and an SEL pattern detection algorithm can be found in [6].

## II. Semantic issues of event pattern languages and motivation of SEL

We motivate SEL by discussing the semantic issues of event pattern languages and related work. The discussion is organized around the event operators in existing languages that we think are problematic and around the temporal specification aspect. Examples used in the discussion are taken from a few representative languages including GEM, ODE, CEDAR, and EBBM. GEM [7] is a declarative rule-based interpreted language designed for real-time distributed system monitoring. ODE [8] is proposed as a specification language for events that are used to fire active database triggers. CEDAR [9], [10] is designed for active databases that are used for network management and therefore has many features targeting at network management applications. EBBM [11] tries to provide high-level abstraction of heterogeneous system behaviors for debugging purposes. But before we start our discussion of the language issues, we first need to define a few terms.

### A. Event model and definition of terms

An *event* or *event instance* is an instantaneous happening of interest. It is defined as a tuple: $< event\ type, event\ time, attribute\ list >$. *Event type* defines the name of a set of events which share a common system defined meaning. *Event time* is the time at which the event occurs. *Attribute list* is a list of typed values which carry further information about the event.

A *primitive event* is an instance of any basic event type defined in a particular event correlation system. Event patterns are expressed as *event expressions*. When an event stream is inspected to detect an event pattern, we say that the corresponding event expression is *evaluated* in the context of the same event stream. A *composite event* $E$ is *fired* when the event pattern $E$ is detected. Each event expression effectively defines a new event type whose instances are all composite events that the event expression can potentially fire.

### B. The negation operator

The semantics of the negation operator (symbolized here as !) as defined in regular expressions are well known[1]. However, it turns out that designing a flexible negation operator suitable for event expressions is not such an easy task. Some languages like EBBM omit the negation operator altogether, losing a powerful

[1] A string matches the regular expression $!E$ if it does not match $E$.

means to express a large class of event patterns. In the context of event pattern detection, it makes more sense to associate the negation operator semantics with time constraints. A straightforward approach is to define $!e$ as a composite event that happens when $e$ never occurs in a specified time interval. Now how this time interval should be specified becomes an issue. GEM requires the negation expression to be delimited by two ordered events as in $\{a;b\}!e$ — $a$ is followed by $b$ with no $e$ happening in between. CEDAR uses a similar approach: in the expression $e$ `not_in` $[a,b]$, $a$ and $b$ are the delimiting events. One problem is that both languages only allow the end of the time interval to be relative to the start of the interval, but they don't allow the start of the time interval to be relative to the end of the interval. If we want to detect $f$ which is not preceded by $e$ in 2 minutes, we would like to specify the pattern using a sliding time window the end point of which is relative to the time of $e$.

An even serious problem is that GEM and CEDAR's negation operators are inflexible. This becomes apparent when one tries to specify a pattern such as: $a$ is followed by $b$ with $f$ but no $e$ happening in between. One can specify the pattern in GEM as $\{a;f;b\}$ & $\{a;b\}!e$ plus an attribute clause constraining the two $a$'s and two $b$'s to be identical respectively. This design is unsatisfactory. The problem is due to the inflexible fashion in which the negation operators are tied to the other operators (the followed-by operator in GEM and the interval construct in CEDAR).

In our language SEL, we define ! in such a way that although the negation sub-expressions are ultimately bound by time intervals, these intervals can be flexibly specified and it is permissible to mix the negation operator with other operators. For example, the aforementioned patterns can be simply expressed as $a \to (f\&!e) \to b$ and $(!e \to f)$ `in 2 minutes`. Our construction is apparently not only more expressive but also very intuitive.

### C. The followed-by operator and the repetition operator

Composite events have durations. For instance, a composite event for $a \to b$ has a duration which starts at primitive event $a$ and ends at primitive event $b$. We say two events *overlap* if their durations overlap. We say two time instants are *contiguous* or one *immediately follows* the other if there is no relevant event happening between them. Note the term "relevant": we do not demand that there be no event of any kind happening in between, but that the events happening in between should be of no relevance to the pattern being detected. In other words, we don't want to reject the actual matching based on the occurrences of events which are of no relevance to the pattern we are detecting. The reason is obvious because only by defining "contiguous" this way can we express event patterns efficiently. With these definitions, the followed-by operator in $E \to F$ may have any of the four semantics described below:

- Non-overlapping, non-contiguous semantics:
$E$'s end time must be earlier than $F$'s start time; other relevant primitive events could happen between the two times. Examples are ODE's *relative*, and EBBM's $\bullet$(*Sequential*).
- Overlapping, non-contiguous semantics:
$E$'s end time must be earlier than $F$'s end time, but there is no constraint for the two start times. Other relevant primitive events could happen between the two end times. Examples are GEM's ';', CEDAR's *fby*, and ODE's *prior*.
- Non-overlapping, contiguous semantics:
$E$'s end time must be earlier than $F$'s start time; no relevant primitive event is allowed to happen between the two times. There is no event language to our knowledge that has this semantics. However, if we think of regular expressions as a very simple event language with the set of relevant events defined as the whole alphabet, its *concatenation* operator has the exact same semantics.
- Overlapping, contiguous semantics:
$E$'s end time must be earlier than $F$'s end time, but there is no constraint for the two start times. No relevant primitive event is allowed to happen between the two end times. Examples are ODE's *sequence*, and CEDAR's *seq*.

Since a repetition expression can be seen as shorthand for a sequence of followed-by expressions, the repetition operator also may have four corresponding semantics based on the followed-by semantics listed above.

For the sake of semantic completeness, it is our opinion that all four semantics should be incorporated. However, there is the dilemma of providing more semantics versus minimizing the number of operators. Our approach to the problem is to provide basic operators which have the potential of easily expressing all semantics in straightforward ways. In our language, we simply provide two operators, $\to$ and $^*$, for followed-by and repetition respectively, and assign them only the "non-overlapping, non-contiguous" semantics. We allow other semantics through easy operator combinations.

The "contiguous semantics" (or "immediately followed-by semantics") is realized by using the ! operator. We can specify "$a$ is immediately followed-by $b$" as $a \to !E \to b$, where $E$ is of the form $e_1|e_2|\ldots$ which represents a list of events that should not happen between $a$ and $b$.

The "overlapping semantics" is realized by converting a composite event to a primitive event through the use of $\{\}$, the *primitive event converter* operator. $\{E\}$ in effect changes the composite event $E$ to a primitive event. For example, we can use $\{a \to b\} \to \{c \to d\}$ to specify $(a \Rightarrow b) \Rightarrow (c \Rightarrow d)$, where $\Rightarrow$ denotes "overlapping, non-contiguous" followed-by. $\{\}$ is also useful in the situation where the duration of a sub-composite event should not have any effect on the composite event. Moreover, use of $\{\}$ can be an effective tool in a distributed environment where a composite event is regarded as having no difference from a primitive event when it is received at a remote site.

Compared to the approach that constrains the timing relationship through attribute predicates, our approach has the advantage of allowing users to figure out the timing constraints by just looking at the major part of the event expression without looking at the attribute predicates. This approach therefore results in more readable expressions.

## D. Temporal relationship specification

In network management, event correlation is traditionally handled as an aggregation procedure over sets of alarms [12], [13], [14]. A typical event correlation rule would be "if we see alarms *a*, *b*, and *c*, all occurring within 30 seconds, the condition is met that indicates a certain fault might have happened." These approaches usually provide little support for specifying timing relationships among events. If temporal constraints must be specified, they are dealt with in the attribute predicate part or the action part of the rules, whereby the event pattern specification is scattered, making the rule harder to understand and maintain. Newer approaches have added better temporal specification capabilities. However, there are a few problems:

- The languages are too bulky with the new capability. For instance, [1] has introduced 9 temporal operators that can be used to specify the relative temporal locations of two events. The utility of some of these operators, such as the "exact" relationship operators "STARTS", "FINISHES", and "COINCIDES", is doubtful since such relationships are based on exact timing which is generally hard to obtain in a distributed environment.
- Limited temporal specification capability. One particular limitation is the flexibility in manipulating time windows. For instance, in many languages, only one time window is allowed in a pattern specification. A pattern that requires multiple windows must be specified, like in [7], by combining multiple rules. This makes it hard to specify situations that require multiple window constraints, which is a useful capability in network management. We believe a language can create more readable rules if it allows such rules to be directly specifiable in single syntactic constructs. Such reasoning is based on the "encapsulation principle" of software engineering.
- Readability of a specification suffers greatly with the introduction of temporal specification capability, particularly when it relies on event attribute semantics. [15] describes an event correlation approach which specifies temporal relationships of events solely through predicates on timing attributes of events. For instance, the following pattern is used to cause an emergency alert to be sent to the network administrator [15]: "event *linkdown* is received but *linkup* is not received within 2 minutes after that, and *alert* has not been generated in the past 5 minutes". To specify this pattern, [15] uses (with other detail omitted):

$@(LinkADown, i) + 2\,\texttt{minutes} \leq @r(LinkUp, @(LinkADown, i), 1)$

$\texttt{and}\ @(LinkADown, i) + 2\,\texttt{minutes} \leq @r(LinkADownAlert, @(LinkADown, i), 1)$

$\texttt{and}\ @(LinkADown, i) + 2\,\texttt{minutes} \geq @r(LinkADownAlert, @(LinkADown, i), 0).$

A specification with our language SEL is much simpler and more intuitive, and therefore more readable:

$!LinkADownAlert \rightarrow \{LinkADown \rightarrow !LinkAUp\ \texttt{in}\ 2\,\texttt{minutes}\}\ \texttt{in}\ 5\,\texttt{minutes}$

## III. SEL SYNTAX AND SEMANTICS

### A. SEL syntax

The syntax of SEL is defined using BNF notation with one exception: $\lfloor$ and $\rfloor$ are used as meta-symbols denoting optional structures, whereas $[$ and $]$ are used as terminals of the language, not BNF meta-symbols. Two types of time window constraints are allowed: a *fixed window* is given by its starting time and finishing time; a *sliding window* is given by an integer number denoting its duration. Details are omitted for *predicate_on_attributes* and *calendar_time*.

$E ::= primitive\_event$

$\quad \| \ E\ \lfloor attribute\_constraint \rfloor\ \lfloor time\_window\_constraint \rfloor$

$\quad \| \ (E) \ \| \ \{E\} \ \| \ E^*n \ \| \ E\ \Delta\ time\_span \ \| \ !E \ \| \ E\ \&E \ \| \ E \mid E \ \| \ E \rightarrow E$

$attribute\_constraint ::= \textbf{if}\ predicate\_on\_attributes$

$time\_window\_constraint ::= \textbf{at}\ [calendar\_time, calendar\_time]$

$\quad \| \ \textbf{in}\ time\_span$

$time\_span ::= n\ \textbf{seconds} \ \| \ n\ \textbf{minutes} \ \| \ n\ \textbf{hours} \ \| \ n\ \textbf{days}$

$n ::= integer$

### B. Composite event semantics

Semantics of composite events delimited by sliding windows are much more difficult to define than fixed windows. We therefore start with the fixed window case. Limited by space, we do not describe the semantics in details including the evaluation of attribute constraints. Interested readers may refer to [6] for more details. However, two important attributes, *starttime* and *time*, are clearly defined because they are indispensable to our approach of semantic specification. These denote the start and end time of a composite event.

#### B.1 Fixed window semantics

The definition is recursively given. All event expressions are delimited by the fixed window constraint `at` $[t_1, t_2]$ which is not repeated below. *CE* denotes a composite event for the corresponding event expression.

- Primitive event *pe*: *CE* happens when a *pe* happens within the fixed window. $starttime(CE) \overset{\text{def}}{=} time(pe)$; $time(CE) \overset{\text{def}}{=} time(pe)$.
- $E\ \Delta\ time\_span$: *CE* happens *time_span* time after an *E* happens provided $time(E) + time\_span$ is inside the fixed window. $starttime(CE) \overset{\text{def}}{=} time(E) + time\_span$; $time(CE) \overset{\text{def}}{=} time(E) + time\_span$.
- $\{E\}$: This is the same as $E\ \Delta\ 0$ `seconds`.
- $!E$: *CE* happens if no *E* happens inside the fixed window. $starttime(CE) \overset{\text{def}}{=} t_1$; $time(CE) \overset{\text{def}}{=} t_2$.
- $E^*n$: *CE* happens when a sequence of at least *n* non-overlapping *E*'s happen within the fixed window. A *CE* is fired for each consecutive sub-sequence of exact *n E*'s. If the sub-sequence is $E_i, \ldots, E_{i+n-1}$, we have: $starttime(CE) \overset{\text{def}}{=} starttime(E_i)$; $time(CE) \overset{\text{def}}{=} time(E_{i+n-1})$.
- $E \mid F$: *CE* happens when an *E* happens within the fixed window; similarly, a *CE* happens when an *F* happens within the fixed window. Note that this implies that one *CE* happens for each instance of *E* and/or *F*. Let *X* represent either *E* or *F*, whichever actually happens, we have: $starttime(CE) \overset{\text{def}}{=} starttime(X)$; $time(CE) \overset{\text{def}}{=} time(X)$.
- $E\&F$: *CE* happens when both an *E* and an *F* happen inside the fixed window. $starttime(CE) \overset{\text{def}}{=} min(starttime(E), starttime(F))$;

$time(CE) \overset{\text{def}}{=}$
$max(time(E), time(F))$.

- $E \rightarrow F$: This definition is more involved and interested readers may refer to [6] for more details:

 1. If $E$'s front-end is "rigid", that is if $starttime(E)$ can be independently determined, then $CE$ happens when an $E$ happens within the fixed window $[t_1, t_2]$, and an $F$ happens within the fixed window $[time(E), t_2]$.

 2. If $E$'s front-end is "fluid", then $F$'s back-end must be "rigid" since we don't allow both these facing ends to be "fluid", $CE$ happens when an $F$ happens within the fixed window $[t_1, t_2]$, and an $E$ happens within the fixed window $[t_1, starttime(F)]$.
$starttime(CE) \overset{\text{def}}{=} time(E)$; $time(CE) \overset{\text{def}}{=} time(F)$.

### B.2 Sliding window semantics

If a composite event $CE$ ever happens and if $CE$ has a rigid front-end, we can detect $CE$ given a fixed window which starts at $starttime(CE)$. Therefore, for this particular event, we can see the sliding window as a fixed window which starts at $starttime(CE)$. In order to determine $starttime(CE)$, we need a new functions: $CE_{first}$, which is a primitive event derived from $CE$ and is used to "mark the start" of a composite event. For example, $(a \rightarrow !b \rightarrow c)_{first}$ is simply $a$. Therefore a $CE$ happens only when a $CE_{first}$ happens, and it is always the case that $starttime(CE) = starttime(CE_{first})$. $CE_{last}$ can be similarly defined with obvious meaning. Using these functions, the semantics for the sliding window case can be easily defined. In the following, event expressions are delimited by the fixed window constraint `in` $sw$.

- If $E$'s back-end is "rigid", we use the fixed window semantics with these two modifications:

 1. For each instance of $CE_{first}$ that happens within the time $sw$, we get a fixed window:
$[starttime(CE_{first}), starttime(CE_{first}) + sw)$.

 2. A $CE$ detected in this fixed window must share the same constituent events as the corresponding $CE_{first}$.

- If $E$'s front-end is "rigid", we use the fixed window semantics with these two modifications:

 1. For each instance of $CE_{last}$ that happens within the time $sw$, we get a fixed window:
$(time(CE_{last}) - sw, time(CE_{last})]$.

 2. A $CE$ detected in this fixed window must share the same constituent events as the corresponding $CE_{last}$.

- If both ends of $E$ are fluid, the expression is illegal.

## C. Examples

The following examples have been used to motivate SEL design. Here we use them to illustrate SEL usage with emphasis on followed-by and negation operators and sliding time windows.

1. Event $a$ happens but $b$ does not happen within 20 seconds after $a$: $a \rightarrow !b$ `in 20 seconds`

2. Event $a$ happens but $b$ does not happen within 5 minutes before $a$: $!b \rightarrow a$ `in 5 minutes`

3. Within 40 seconds, event $a$ happens followed by $b$, but event $c$ does not happen in between: $a \rightarrow !c \rightarrow b$ `in 40 seconds`

4. Report at the first instance of event $b$ after event $a$ within 2 minutes: $a \rightarrow !b \rightarrow b$ `in 2 minutes`

5. Event $a$ happens but $b$ does not happen within 2 minutes, and event $c$ did not happen in the past 5 minutes:
$!c \rightarrow (\{a \rightarrow !b\}$ `in 2 minutes`$)$ `in 5 minutes`

6. Event $a$ happens but is not preceded by $b$ in the previous 20 seconds; $a$ is then followed by $c$ and $d$ or followed by $c$ but not $e$ in the next 2 minutes:
$(!b \rightarrow a$ `in 20 seconds`$) \rightarrow (c \ \& \ (d \mid !e)$ `in 2 minutes`$)$

7. Four or more successive US Federal Reserve rate cuts ($c$) without an intervening rate increase ($i$) in 2 years:
$c \rightarrow \ !i \rightarrow c \rightarrow \ !i \rightarrow c \rightarrow \ !i \rightarrow c$ `in 730 days` or
$(c \rightarrow \ !i)^*[3,3] \rightarrow c$ `in 730 days`

## References

[1] G. Jakobson and M. Weissman. Real-Time Telecommunication Network Management: Extending Event Correlation with Temporal Constraints. In Sethi et al. [16], pages 291–301.

[2] S. Yemini, S. Kliger, E. Mozes, Y. Yemini, and D. Ohsie. High Speed and Robust Event Correlation. *IEEE Communications*, May 1996.

[3] R.D. Gardner and D.A. Harle. Pattern Discovery and Specification Techniques for Alarm Correlation. In *Proceedings of the IEEE/IFIP 1998 Network Operations and Management Symposium*, pages 713–722, New Orleans, Louisiana, USA, February 1998. IEEE.

[4] P. Kalyanasundaram, A. Sethi, C. Sherwin, and D. Zhu. A Spreadsheet-Based Scripting Environment for SNMP. In A. Lazar, R. Saracco, and R. Stadler, editors, *Integrated Network Management V*, pages 752–765. London: Chapman & Hall, 1997.

[5] D. Zhu and A. Sethi. Debugging Network Management Scripting Applications. Submitted for publication.

[6] D. Zhu and A. Sethi. SEL, A New Event Pattern Specification Language for Network Management Event Correlation. CIS Technical Report 2002-01, University of Delaware, July 2001.

[7] M. Mansouri-Samani and M. Sloman. GEM: A Generalized Event Monitoring Language for Distributed Systems. *IEE/IOP/BCS Distributed Systems Engineering Journal*, 4(2):96–108, June 1997.

[8] N. Gehani, H.V. Jagadish, and O. Shumeli. Composite Event Specification in Active Databases: Model and Implementation. In *Proc. 18th International Conference on Very Large Data Bases*, pages 100–111, Vancouver, Canada, 1992.

[9] M. Hasan. An Active Temporal Model for Network Management Databases. In Sethi et al. [16], pages 524–535.

[10] M. Hasan. *The Management of Data, Event, and Information Presentation for Network Management*. PhD thesis, University of Waterloo, Waterloo, Canada, 1996.

[11] P. Bates. Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behavior. *ACM Transactions on Computer Systems*, 13(1):1–31, February 1995.

[12] S. Brugbosi, G. Bruno, and et al. An Expert System for Real-Time Fault Diagnosis of the Italian Telecommunications Network. In H. Hegering and Y. Yemini, editors, *Integrated Network Management III*, pages 617–628. North Holland, Amsterdam, 1993.

[13] G. Jakobson and M. Weissman. Alarm Correlation. *IEEE Network*, 7(6), November 1993.

[14] K. Appleby, G. Goldszmidt, and M. Steinder. Yemanja - a Layered Event Correlation Engine for Multidomain Server Farms. In G. Pavlou, N. Anerousis, and A.Liotta, editors, *Integrated Network Management VII*, pages 329–344. Piscataway, NJ: IEEE Publishing, 2001.

[15] G. Liu, A.K. Mok, and E.J. Yang. Composite Events for Network Event Correlation. In Sloman et al. [17], pages 247–260.

[16] A. Sethi, Y. Raynaud, and F. Faure-Vincent, editors, *Integrated Network Management IV*. London: Chapman & Hall, 1995.

[17] M. Sloman, S. Mazumdar, and E. Lupu, editors, *Integrated Network Management VI*. Piscataway, NJ: IEEE Publishing, 1999.