An Interval-Based Algebra for Restricted Event Detection

Jan Carlson and Björn Lisper

Department of Computer Science and Engineering
Mälardalen University, Sweden
jan.carlson@mdh.se bjorn.lisper@mdh.se

Abstract. In this article, we propose an interval based algebra for detection of complex events. The algebra includes a strong restriction policy in order to comply with the resource requirements of embedded or real-time applications. We prove a set of algebraic properties to justify the novel restriction policy and to establish the relation between the unrestricted algebra and the restricted version. Finally, we present an efficient algorithm that implements the proposed algebra.

1 Introduction

A wide range of applications, including active databases, traffic monitoring systems and rule based embedded systems, are based on the detection of events that trigger an appropriate response from the system. Events can be simple, e.g., sampled directly from the environment or occuring within the system, but it is often necessary to react to more sophisticated situations involving a number of simpler events that occur in accordance with some pattern.

A standard way in which to allow systems to react to sophisticated situations is to introduce complex events by means of an event algebra. These complex events can then be used to trigger actions just like simple events. A benefit of this method is that the mechanisms handling event detection are separated from the rest of the system logic.

Since our primary interest concerns embedded applications and systems with strict timeliness requirements, it is essential that the event detection can be implemented with limited resources. As a result, the algebra must be restricted so as to only detect a subset of all possible occurrences of complex events. This can be achieved by applying a suitable restriction policy, as will be described in the next section.

A great many event algebras have been proposed for different applications. Most of them include operators such as disjunction, sequence, conjunction and some form of negation, but the semantics of these operators vary. Further, many systems add to these some operators of their own. Restriction policies are typically informally defined and little effort spent determining the algebraic properties of the algebra.

We propose an interval based event algebra with well-defined formal semantics, and with a restriction policy strong enough to make it effectively implementable. We also state a number of algebraic properties, including a clear description of the relation between the unrestricted algebra and the restricted version. Finally, we present an efficient algorithm that implements the proposed algebra.

The rest of this paper is organised as follows: Section 2 introduces techniques commonly used in event algebras and presents related work. The algebra is defined in Section 3, followed by a presentation of the algebraic properties in Section 4. Section 5 presents the algorithm, and Section 6 concludes the paper.

2 Event Algebras

The following operations, or variants of them, are found in most event algebras. *Disjunction* of A and B means that either of A and B occurs, here denoted $A \vee B$. *Conjunction* means that both events have occurred, possibly not simultaneously, and is denoted $A + B$. The *negation*, denoted $A - B$, occurs when there is an occurrence of A during which there is no occurrence of B. Finally, a *sequence* of A and B is an occurrence of A followed by an occurrence of B, and is denoted $A; B$.

Examples of how event algebras are used in the area of active datebases include SAMOS [5], Snoop [3] and Ode [6]. These three systems differ primarily in the choice of detection mechanism. SAMOS is based on Petri nets, while Snoop uses event graphs. In Ode, event definitions are equivalent to regular expressions and can be detected by state automata.

A formalized schema for this type of event detection, including a definition of the operations and restriction policies of Snoop using this schema, has been defined by Mellin and Andler [10]. Liu et al. uses Real Time Logic to define the semantics of an event detection system. As a result, the conditions for event occurrences can be transformed into timing constraints and handled by general timing constraint monitoring techniques [9].

The event algebra developed by Hinze and Voisard is designed to suite event notification service systems in general [7]. Their algebra contains time restricted sequence and conjunction, which permits events like *A occurs less than t time units before B* to be expressed.

In the area of knowledge representation, similar techniques are used to reason about event occurrences. Interval Calculus introduce formalised concepts for properties, actions and events, where events are expressed in terms of conditions for their occurrence [2]. Event Calculus [8] also deals with the occurrences of events, but, as in the Interval Calculus, the motivation is slightly different from ours. Rather than detecting complex events as they occur, the focus of Event Calculus is the inferences that can be made from the fact that certain events have occurred.

2.1 Restricted Detection

A very straightforward definition of the sequence operator is that the sequence $A;B$ should occur whenever A occurs and then B occurs. Using this definition, three occurrences of A followed by two occurrences of B would generate six occurrences of the sequence. While this may be acceptable, or even desirable, in some applications, the memory requirements (each occurrence of A must be remembered forever) and the increasing number of simultaneous events means that it is unsuitable in many cases. Also, it is argued that many applications are interested only in a subset of the instances that are generated by this definition.

One way to deal with this is to define the event algebra in two steps. The operations are defined in an unrestricted, straightforward way like in our example above. Then a restriction policy is defined. This acts like a filter, so that only a subset of the occurrences allowed by the unrestricted definition are detected. For example, the restriction policy could state that only the latest occurrence of A are allowed to create an occurrence of $A;B$ when B occurs.

This type of restriction based methods are for example used by Snoop [3] and in the algebra proposed by Hinze and Voisard [7]. Zimmer and Unland present a formal restriction framework in which the event algebras of Snoop, SAMOS and Ode are compared [11].

2.2 Interval-Based Event Detection

Single point detection means that every complex event, including those that occur during a time interval, is associated with a single time point (the time of detection, i.e, the end of the occurrence interval). Galton and Augusto [4] showed that this results in unintended semantics for some operation compositions.

For example, using single point detection an instance of the event $A;(B;C)$ is detected if B occurs first, and then A followed by C. The reason is that these occurrences cause a detection of $B;C$ which is associated with the occurrence time of C. Since A occurs before this time point, an occurrence of $A;(B;C)$ is detected.

This problem can be solved by associating the occurrence of a complex event with the occurrence interval, rather than the time of detection. In this setting, the sequence $A;B$ can be defined to occur only if the intervals of A and B are non-overlapping. In our example, no occurrence of $A;(B;C)$ would be detected, since A occurs within the interval associated with the occurrence of $B;C$.

Most event algebras, especially in the area of active databases, use single point detection. An interval based version of Snoop has been developed by Adaikkalavan and Chakravarthy [1], and the work by Mellin and Andler is also based on intervals [10].

3 The Event Algebra

The system is assumed to have a pre-defined set of primitive events that it should be able to react to. These events can be external (sampled from the environment

or originating from another system) or internal (such as the violation of a condition over the system state, or a timeout), but the detection mechanism does not distinguish between these categories.

We assume occurrences of primitive events to be instantaneous and atomic, and allow occurrences to carry values. This value could for example identify at which external device the event occurred, or be some measured value from the environment. The values are not manipulated in any way by the detection mechanism, but simply forwarded to the part of the system that reacts to the detected events. An occurrence of a primitive event is represented by the tuple $\langle \upsilon, \tau \rangle$, where υ is the value (υ belongs to some arbitrary domain of values), and τ is the time of the occurrence. We assume a discrete time modelled by the natural numbers.

3.1 Basic Concepts

From the simple events, represented by a set $\mathcal{I}$ of identifiers, expressions representing complex events can be constructed as follows.

Definition 1. *Given a set $\mathcal{I}$ of identifiers we define:*

- *If $A \in \mathcal{I}$, then A is an event expression.*
- *If A and B are event expressions, so are $A \vee B$, $A + B$, $A - B$ and $A; B$.*

The complex event expressions in the definition represent disjunction, conjunction, negation and sequence, respectively.

Definition 2. *An* event instance *is a set of value-time tuples. A* primitive event instance *is a singleton set. For an event instance a, we define:*

$$\begin{aligned} start(a) &= Min_{\langle \upsilon, \tau \rangle \in a}\ (\tau) \\ end(a) &= Max_{\langle \upsilon, \tau \rangle \in a}\ (\tau) \end{aligned}$$

From the definition follows that for any primitive event instance a, $start(a) = end(a)$. Non-primitive event instances are considered to occur throughout an interval from the earliest of the included primitive event instances, to the latest one.

All instances of a particular event form an event stream. The semantics of the algebra, presented below, associates with each event expression a corresponding event stream.

Definition 3. *An* event stream *is a set of event instances. An event stream A is said to be* non-simultaneous *if all instances have different end times. A* primitive event stream *is a non-simultaneous event stream containing only primitive event instances.*

3.2 Unrestricted Semantics

Definition 4. *For an event stream S and an event instance a, define $empty(S, a)$ to hold iff there is no $s \in S$ such that $start(a) \leq start(s)$ and $end(s) \leq end(a)$.*

The following four functions over event streams form the core of the algebra semantics, as they define the basic characteristics of the four operations.

Definition 5. *For event streams S and T, define:*

$$\begin{aligned} dissem(S,T) &= S \cup T \\ consem(S,T) &= \{s \cup t \mid s \in S \wedge t \in T\} \\ negsem(S,T) &= \{s \mid s \in S \wedge empty(T,s)\} \\ seqsem(S,T) &= \{s \cup t \mid s \in S \wedge t \in T \wedge end(s) < start(t)\} \end{aligned}$$

Definition 6. *An* interpretation *is a function that maps each identifier in $\mathcal{I}$ to a primitive event stream.*

Definition 7. *The unrestricted meaning of an event expression for a given interpretation $\mathcal{S}$ is defined as follows:*

$$\begin{aligned} [A]^{\mathcal{S}} &= \mathcal{S}(A) \text{ if } A \in \mathcal{I} \\ [A \vee B]^{\mathcal{S}} &= dissem([A]^{\mathcal{S}}, [B]^{\mathcal{S}}) \\ [A + B]^{\mathcal{S}} &= consem([A]^{\mathcal{S}}, [B]^{\mathcal{S}}) \\ [A - B]^{\mathcal{S}} &= negsem([A]^{\mathcal{S}}, [B]^{\mathcal{S}}) \\ [A; B]^{\mathcal{S}} &= seqsem([A]^{\mathcal{S}}, [B]^{\mathcal{S}}) \end{aligned}$$

To simplify the presentation, we will use the notation $[A]$ instead of $[A]^{\mathcal{S}}$ whenever the choice of $\mathcal{S}$ is obvious or arbitrary.

3.3 Restricted Semantics

As discussed in the introduction, due to efficiency considerations we have to restrict the detection to a subset of the instances defined by the unrestricted semantics. As a first step, we remove simultaneous instances of an event stream (i.e., instances a and a' of the same event stream with $end(a) = end(a')$). In order not to lose the desired algebraic properties, this filtering must be done carefully.

Definition 8. *Let remsim be any function over event streams such that the following holds. For an event stream S, $remsim(S)$ is a minimal subset of S such that for any element $s \in S$ there is an element $s' \in remsim(S)$ with $start(s) \leq start(s')$ and $end(s) = end(s')$.*

Informally, from a number of instances with the same end time, we keep only one with maximal start time. Using discrete time ensures that such a function exists.

For all operations except sequence, this restriction is enough to allow an efficient implementation (negation does not need any restriction at all). For

sequence, however, we also have to deal with the problem that in the unrestricted version, each occurrence of the first argument is used over and over again in combination with all subsequent instances of the second argument. This means that every instance of the first argument must be stored throughout the system lifetime, thus precluding an implementation with limited resources.

Definition 9. *Let restrict be any function over event streams such that the following holds. For an event stream S, restrict(S) is a minimal subset of S such that for any element $s \in S$ there is an element $s' \in$ restrict(S) with $start(s) \leq start(s')$ and $end(s') \leq end(s)$.*

Informally, when detecting a sequence $A;B$, an instance of A can only be combined with the earliest possible instance of B. Similarly, an instance of B can only be combined with the latest possible instance of A. This is similar, but not equivalent, to the recent context of Snoop.

Definition 10. *The restricted meaning of an event expression for a given interpretation $\mathcal{S}$ is:*

$$\begin{aligned}
[\![A]\!]^{\mathcal{S}} &= \mathcal{S}(A) \text{ if } A \in \mathcal{I} \\
[\![A \vee B]\!]^{\mathcal{S}} &= remsim(dissem([\![A]\!]^{\mathcal{S}}, [\![B]\!]^{\mathcal{S}})) \\
[\![A + B]\!]^{\mathcal{S}} &= remsim(consem([\![A]\!]^{\mathcal{S}}, [\![B]\!]^{\mathcal{S}})) \\
[\![A - B]\!]^{\mathcal{S}} &= negsem([\![A]\!]^{\mathcal{S}}, [\![B]\!]^{\mathcal{S}}) \\
[\![A;B]\!]^{\mathcal{S}} &= restrict(seqsem([\![A]\!]^{\mathcal{S}}, [\![B]\!]^{\mathcal{S}}))
\end{aligned}$$

As in the unrestricted version, we will use the notation $[\![A]\!]$ instead of $[\![A]\!]^{\mathcal{S}}$ whenever the choice of $\mathcal{S}$ is obvious or arbitrary.

Example 1. To illustrate the difference between the unrestricted and the restricted semantics, these tables show the event instances of A and B (which we assume to be primitive, so $[A] = [\![A]\!]$ and $[B] = [\![B]\!]$), together with the corresponding instances of the complex events $A+B$ and $A;B$, using both unrestricted and restricted semantics.

Expression	Instances
$[A]$	
$[B]$	
$[A + B]$	
$[\![A + B]\!]$	

Expression	Instances
$[A]$	
$[B]$	
$[A;B]$	
$[\![A;B]\!]$	

4 Algebraic Properties

A main concern regarding the restriction policy has been to ensure that the restricted algebra should comply with the algebraic laws that intuitively should

hold for an event algebra. Disjunction and sequence should be associative, conjunction should be distributive over disjunction, etc. This is not the only requirement, however, since it would be trivially satisfied by a restriction policy that simply filters away all instances. The restriction policy should remove as few instances as possible, while still ensuring the desired algebraic properties and allowing an implementation with bounded resources. More specifically, we want a theoretical description of the relation between the unrestricted semantics and the restricted version.

The following theorem justifies the proposed restriction policy. The subset result is not trivial, since with a different restriction policy $[\![B]\!] \subset [B]$ could easily mean that $[\![A - B]\!] \supset [A - B]$. The second statement ensures that our restriction policy does not remove too much. Every removed instance leaves some trace in the restricted version, as the interval between the start and end time of the removed instance must be non-empty.

Theorem 1. *For any event expression A, the following holds:*

i) $[\![A]\!] \subseteq [A]$
ii) $a \in [A] \;\Rightarrow\; \exists_{a' \in [\![A]\!]}\ (start(a) \leq start(a') \wedge end(a') \leq end(a))$

Proof. We prove the theorem by structural induction over expressions. As a base case, both statements hold trivially for any primitive event expression since $[\![A]\!] = [A]$ when $A \in \mathcal{I}$. For the inductive case, assume that both statements hold for event expressions A_1 and A_2. From Definition 5, and the fact that $restrict(P) \subseteq P$ and $remsim(S) \subseteq S$, it follows that statement *i)* holds for $A_1 \vee A_2$, $A_1 + A_2$ and $A_1; A_2$.

In order to show that statement *i)* holds for negation, take an arbitrary $a \in [\![A_1 - A_2]\!]$. Then $a \in [\![A_1]\!]$ and $empty([\![A_2]\!], a)$. By assumption *i)*, this means that $a \in [A_1]$ and assumption *ii)* implies $empty([A_2], a)$. Thus, $a \in negsem([A_1], [A_2])$, so $a \in [A_1 - A_2]$ which means that we have $[\![A_1 - A_2]\!] \subseteq [A_1 - A_2]$.

Continuing the inductive case with statement *ii)*, we consider first the case of sequence. We take an arbitrary $a \in [A_1; A_2]$ which implies $a = a_1 \cup a_2$ where $a_1 \in [A_1]$ and $a_2 \in [A_2]$ with $end(a_1) < start(a_2)$. By assumption *ii)*, there are instances $a_1' \in [\![A_1]\!]$ and $a_2' \in [\![A_2]\!]$ such that $start(a_i) \leq start(a_i')$ and $end(a_i') \leq end(a_i)$ for $i \in \{1, 2\}$ and thus $a_1' \cup a_2' \in seqsem([\![A_1]\!], [\![A_2]\!])$. Then, by the definition of *restrict*, there must be some element $a' \in restrict(seqsem([\![A_1]\!], [\![A_2]\!]))$ with $start(a_1') \leq start(a')$ and $end(a') \leq end(a_2')$. So, we have found an instance $a' \in [\![A_1; A_2]\!]$ for which $start(a) = start(a_1) \leq start(a_1') \leq start(a')$ and $end(a') \leq end(a_2') \leq end(a_2) = end(a)$.

For negation, we take an arbitrary $a \in [A_1 - A_2]$. This implies $a \in [A_1]$ and $empty([A_2], a)$, which by assumption *i)* means that $empty([\![A_2]\!], a)$. By assumption *ii)*, there is an instance $a' \in [\![A_1]\!]$ with $start(a) \leq start(a')$ and $end(a') \leq end(a)$. We have $empty([\![A_2]\!], a')$, and thus $a' \in negsem([\![A_1]\!], [\![A_2]\!])$. So, we have found an instance $a' \in [\![A_1 - A_2]\!]$ for which $start(a) \leq start(a')$ and $end(a') \leq end(a)$.

The proofs for disjunction and conjunction are similar to the cases above, and have been left out due to space limitations. Together, this proves by induction that both statements hold for any event expression A. □

In order to reason about algebraic properties like associativity, etc. we must define a relaxed concept of equivalence. As a result of the restriction policy, the two sets $[\![A;(B;C)]\!]$ and $[\![(A;B);C]\!]$ are not necessarily equal. However, we can show that for every instance of $[\![A;(B;C)]\!]$ there is an instance of $[\![(A;B);C]\!]$ with the same start- and end time, and vice versa. This means, for example, that in systems where events are used to trigger response actions, the two expressions would trigger actions at the same time (although possibly with different values). This time based notion of equality is formalised as follows.

Definition 11. *For event instances a and b, event streams S and T, and event expressions A and B, define:*

$$\begin{array}{ll} a \cong b & \textit{iff } \mathit{start}(a) = \mathit{start}(b) \textit{ and } \mathit{end}(a) = \mathit{end}(b) \\ S \cong T & \textit{iff } \{\langle \mathit{start}(a), \mathit{end}(a)\rangle \mid a \in S\} = \{\langle \mathit{start}(b), \mathit{end}(b)\rangle \mid b \in T\} \\ A \cong B & \textit{iff } [\![A]\!] \cong [\![B]\!] \end{array}$$

Trivially, $\cong$ in an equivalence relation. Moreover, we will show that it satisfies the substitutive condition, and hence defines structural congruence over event expressions. For the proof, we need the following lemma.

Lemma 1. *For event streams such that $S \cong S'$ and $T \cong T'$, we have:*

$$\begin{array}{lll} \mathit{dissem}(S,T) \cong \mathit{dissem}(S',T') & & \mathit{negsem}(S,T) \cong \mathit{negsem}(S',T') \\ \mathit{consem}(S,T) \cong \mathit{consem}(S',T') & & \mathit{remsim}(S) \cong \mathit{remsim}(S') \\ \mathit{seqsem}(S,T) \cong \mathit{seqsem}(S',T') & & \mathit{restrict}(S) \cong \mathit{restrict}(S') \end{array}$$

Proof. The four equivalences regarding *dissem*, *consem*, etc. follow trivially from the fact that Definition 5 only considers start and end times. For the *remsim* equivalence, take an arbitrary $a \in \mathit{remsim}(S)$. Then $a \in S$ so there is an $a' \in S'$ with $a \cong a'$. The definition of *remsim* implies that there is some $b \in \mathit{remsim}(S')$ such that $\mathit{start}(a') \leq \mathit{start}(b)$ and $\mathit{end}(b) = \mathit{end}(a')$. In the same way, there is a corresponding element $b' \in S$ such that $b \cong b'$ so there is some element $c \in \mathit{remsim}(S)$ with $\mathit{start}(b') \leq \mathit{start}(c)$ and $\mathit{end}(c) = \mathit{end}(b')$.

We have two elements a and c in $\mathit{remsim}(S)$ with $\mathit{start}(a) \leq \mathit{start}(c)$ and $\mathit{end}(a) = \mathit{end}(c)$. Assuming $a \neq c$, the set $\mathit{remsim}(S) - \{a\}$ meets the requirement in the definition of *remsim*, contradicting the minimality. Hence, we must have $a = c$, which implies $\mathit{start}(a) = \mathit{start}(b)$. So, for an arbitrary $a \in \mathit{remsim}(S)$ we have found a $b \in \mathit{remsim}(S')$ with $a \cong b$, and hence $\mathit{remsim}(S) \cong \mathit{remsim}(S')$.

The proof of the *restrict* equivalence is very similar to the one above. □

Theorem 2. *If $A_1 \cong A_1'$ and $A_2 \cong A_2'$ then we have $(A_1 \vee A_2) \cong (A_1' \vee A_2')$, $(A_1 + A_2) \cong (A_1' + A_2')$, $(A_1 - A_2) \cong (A_1' - A_2')$ and $(A_1; A_2) \cong (A_1'; A_2')$.*

Proof. This follows trivially from Lemma 1 and Definition 10. □

Using the weak equivalence, we can formulate a number of algebraic laws.

Theorem 3. *For any event expressions A, B and C, the following laws hold:*

$$
\begin{array}{lrcl}
R1: & A \vee B & \cong & B \vee A \\
R2: & A \vee A & \cong & A \\
R3: & A \vee (B \vee C) & \cong & (A \vee B) \vee C \\
R4: & A;(B;C) & \cong & (A;B);C \\
R5: & A + B & \cong & B + A \\
R6: & A + A & \cong & A \\
R7: & A + (B + C) & \cong & (A + B) + C \\
R8: & A + (B \vee C) & \cong & (A + B) \vee (A + C) \\
R9: & (A \vee B) + C & \cong & (A + C) \vee (B + C) \\
R10: & (A \vee B) - C & \cong & (A - C) \vee (B - C) \\
R11: & (A - B) - B & \cong & A - B \\
R12: & A - (B \vee C) & \cong & (A - B) - C
\end{array}
$$

Proof. $R1$, $R2$ and $R3$ follow trivially from Definitions 10 and 5 and the definition of *remsim*. For $R4$, we first take an arbitrary $d \in [\![A;(B;C)]\!]$. Using Theorem 1 it is straightforward to show that $d \in [(A;B);C]$ which implies that there is some $d' \in [\![(A;B);C]\!]$ with $start(d) \leq start(d')$ and $end(d') \leq end(d)$. In the same way, this implies that there is some $d'' \in [\![A;(B;C)]\!]$ with $start(d') \leq start(d'')$ and $end(d'') \leq end(d')$. The minimality condition in the definition of *restrict* means that we must in fact have $d \cong d''$, which implies $d \cong d'$. Thus, for an arbitrary $d \in [\![A;(B;C)]\!]$ there is a$d' \in [\![(A;B);C]\!]$ such that $d \cong d'$. In the same way we can show that for an arbitrary $d \in [\![(A;B);C]\!]$ there is a $d' \in [\![A;(B;C)]\!]$ with $d \cong d'$.

$R5$ and $R6$ follow trivially from Definitions 10 and 5 and the definition of *remsim*. The proofs of $R7$ and $R8$ are very similar to that of $R4$. $R9$ follows trivially from $R5$ and $R8$.

For $R10$, we take an arbitrary $d \in [\![(A \vee B) - C]\!]$. This means that $d \in [\![A \vee B]\!]$ and $empty([\![C]\!], d)$. Thus either $d \in [\![A]\!]$ or $d \in [\![B]\!]$, which means that $d \in [\![A - C]\!]$ or $d \in [\![B - C]\!]$, but in both cases we have $d \in dissem([\![A - C]\!], [\![B - C]\!])$. Thus there is some $d' \in [\![(A - C) \vee (B - C)]\!]$ with $start(d) \leq start(d')$ and $end(d) = end(d')$. Since $d' \in dissem([\![A]\!], [\![B]\!])$, by minimality of *remsim* we must have $d \cong d'$. In a similar way we can show that any $d \in [\![(A - C) \vee (B - C)]\!]$ implies the existence of an $d' \in [\![(A \vee B) - C]\!]$ such that $d \cong d'$.

$R11$ follows trivially from Definitions 10 and 5. For $R12$, if $a \in [\![A - (B \vee C)]\!]$ we have $a \in [\![A]\!]$ and $empty([\![B \vee C]\!], a)$. By Theorem 1, we must have $empty([B \vee C], a)$ and thus, $empty([\![B]\!], a)$ and $empty([\![C]\!], a)$. Then $a \in [\![A - B]\!]$, and $a \in [\![(A - B) - C]\!]$. Starting instead with an $a \in [\![(A - B) - C]\!]$, this means $a \in [\![A]\!]$, $empty([\![B]\!], a)$ and $empty([\![C]\!], a)$. Then $empty([\![B \vee C]\!], a)$ and thus $a \in [\![A - (B \vee C)]\!]$. □

5 Event Detection Algorithm

For the detection algorithm, we let 1 denote the first time point at which events may occur, using 0 only when referring to the time of system initialisation.

To simplify the algorithm presentation, we use the following auxiliary functions (*match* is not well-defined, but any function that meets the condition can be used).

$$
\begin{array}{lcl}
get(A, \mathcal{S}, \tau) & = & \begin{cases} \{\langle v, \tau \rangle\} \text{ if } \{\langle v, \tau \rangle\} \in \mathcal{S}(A) \\ \langle\rangle \text{ if no such instance exists in } \mathcal{S}(A) \end{cases} \\
match(y, q) & = & \begin{cases} y \cup \text{an element in } filter(y, q) \text{ with maximum start time} \\ \langle\rangle \text{ if } filter(y, q) \text{ is empty} \end{cases} \\
filter(y, q) & = & \{e \mid e \in q \wedge end(e) < start(y)\}
\end{array}
$$

The symbol $\langle\rangle$ is used to represent a non-occurrence, and we use the symbol τ^c when referring to the current time in the algorithm. Since each operator occurrence in the expression requires its own state variables, we simplify the presentation by using variables that are indexed with subexpressions. Thus, for each subexpression A, v_A denotes the v variable of A. An equivalent method would be to number each subexpression, and use ordinary integer indexed variables.

5.1 Algorithm Description

Figure 1 presents the algorithm for detecting an event expression E. The algorithm is presented in a meta format that can be instantiated for any fixed expression. The top level conditionals can be evaluated statically, which permits statically unrolling the foreach statement. All indices can also be evaluated statically. A concrete example of this is given in Example 2.

In the initial state, at time 0, let $w_A = z_A = \langle\rangle$, $t_A = 0$ and $q_A = \emptyset$ for every subexpression A in E. Each time instant, the algorithm takes as input the current instances of primitive events (provided by the *get* function) and computes the current instance of E, if there is one. The following theorem formalises the output of the algorithm.

Theorem 4. *For any subexpression A in E, after executing the algorithm at time instants 1 to τ, $v_A = a$ if there is an instance $a \in [\![A]\!]$ with $end(a) = \tau$. If there is no such instance in $[\![A]\!]$, $v_A = \langle\rangle$.*

Proof. We only outline very informally the core of the correctness proof, providing some intuition to the relation between the algorithm and the formal semantics. When processing a subexpression A on the form $B \vee C$, v_B and v_C already contain the current instances of B and C, respectively, since the original expression is processed bottom-up. The algorithm assigns to v_A the one with latest start time, which according to the definition of *remsim* is the current instance of A.

Conjunctions are handled by storing in w and z the instances with latest start time from B and C, respectively. If there is a current instance of B or

For each subexpression A in E, in bottom-up order, do the following:

if $A \in \mathcal{I}$ then $v_A := get(A, \mathcal{S}, \tau^c)$

if A is $B \vee C$ then
  if $v_B = \langle\rangle$ or ($v_C \neq \langle\rangle$ and $start(v_B) \leq start(v_C)$)
    then $v_A := v_C$
    else $v_A := v_B$

if A is $B + C$ then
  if $v_B \neq \langle\rangle$ and ($w_A = \langle\rangle$ or $start(w_A) < start(v_B)$) then $w_A := v_B$
  if $v_C \neq \langle\rangle$ and ($z_A = \langle\rangle$ or $start(z_A) < start(v_C)$) then $z_A := v_C$
  if $v_B \neq \langle\rangle$ and (($v_C = \langle\rangle$ and $z_A \neq \langle\rangle$) or
                ($v_C \neq \langle\rangle$ and $start(v_C) \leq start(v_B)$))
    then $v_A := v_B \cup z_A$
  if $v_C \neq \langle\rangle$ and (($v_B = \langle\rangle$ and $w_A \neq \langle\rangle$) or
                ($v_B \neq \langle\rangle$ and $start(v_B) < start(v_C)$))
    then $v_A := w_A \cup v_C$

if A is $B - C$ then
  if $v_C \neq \langle\rangle$ and $t_A < start(v_C)$ then $t_A := start(v_C)$
  if $v_B \neq \langle\rangle$ and $t_A < start(v_B)$ then $v_A := v_B$

if A is $B; C$ then
  if $v_C = \langle\rangle$ then $v_A := \langle\rangle$
    else $v_A := match(v_C, q_A)$; $q_A := q_A - filter(v_C, q_A)$
  if $v_B \neq \langle\rangle$ and $t_A < start(v_B)$
    then $q_A := q_A \cup \{v_B\}$; $t_A := start(v_B)$

Fig. 1. Meta-algorithm for the detection of E under the interpretation $\mathcal{S}$

C, the current instance of A is formed by combining instances from B and C such that at least one is a current instance, and such that the start time of the combination is as late as possible.

For negations, the variable t contains the latest start time of all instances of C that has occurred until now. The current instance of B becomes the current instance of A if it starts later than t, which conforms to the definition of *negsem*.

To deal with sequences, the variable q stores instances of B that has not yet been possible to match with any instance of C. In addition, the variable t is used to ensure that no instances in q are fully overlapping. If there is a current instance of C, it is combined with the best matching instance in q to form the current instance of A. Also, the definition of *restrict* dictates that instances of B that end before the start time of the current instance of C, may not be used to form future instances of A. Hence, these are removed from q. □

Example 2. Assume we are detecting the event $(A \vee B) - C$. After instantiating the meta-algorithm, we can unroll the foreach statement and statically evaluate

the top-level conditionals. We also instantiate the subexpression indicies with corresponding integers. The resulting algorithm is presented in Figure 2.

$v_1 := get(A, \mathcal{S}, \tau^c)$
$v_2 := get(B, \mathcal{S}, \tau^c)$
$v_3 := get(C, \mathcal{S}, \tau^c)$
if $v_1 = \langle\rangle$ or ($v_2 \neq \langle\rangle$ and $start(v_1) \leq start(v_2)$)
  then $v_4 := v_2$
  else $v_4 := v_1$
if $v_3 \neq \langle\rangle$ and $t_5 < start(v_3)$ then $t_5 := start(v_3)$
if $v_4 \neq \langle\rangle$ and $t_5 < start(v_4)$ then $v_5 := v_4$

Fig. 2. Instantiated algorithm for detecting $(A \vee B) - C$

6 Conclusions and Future Work

We have developed an interval based algebra for detection of complex events. The algebra includes a strong restriction policy in order to comply with the resource requirements of embedded or real-time applications. The restriction policy is justified by a theorem stating that it never adds instances, compared to the unrestricted semantics. Also, every removed instance leaves some trace in the restricted version, as the interval between the start and end time of the removed instance must still contain at least one instance.

An event detection algorithm that implements the proposed algebra was presented. In this algorithm, each disjunction, conjunction and negation in the event expression requires a constant amount of storage, and contributes with a constant factor to the computation time. For the sequence $A; B$, on the other hand, a set of instances must be stored and the computation time is proportional to the size of this set. This is a result of Theorem 1, since it is not enough to store a single best instance of A (i.e., the one with latest start time). Once an instance of B occurs, it must be combined with the best *allowed* instance of A. This might not be the best instance of A that has occurred so far, if the interval of B is long.

This is clearly a weakness, but as it follows from one of the desired properties of the restriction, we have to look for other ways to ensure limited resource demands. The maximum size of the storage set for $A; B$ depends on the relative frequence of occurrences in A and B. Roughly, if no more that n instances of A can occur during the longest possible interval in which no B occurs, n is the maximum size of the storage set.

We are currently formalising this idea, including how to calculate frequence bounds for complex events from frequence bounds of the primitive events. This

seems to be possible for all expressions except negations, so there is still a problem with expressions like $A;(B-C)$. If C and B occur together, $B-C$ never occurs at all, so every instance of A is stored forever.

Additional future work includes finishing the formal proof of Theorem 4.We are also considering extending the algebra with additional operations, especially time limited versions of sequence and conjunction.

References

1. R. Adaikkalavan and S. Chakravarthy. Event operators: Formalization algorithms, and implementation using interval-based semantics. Technical Report CSE-2002-3, University of Texas at Arlington, Department of Computer Science and Engineering, June 2002.
2. J. F. Allen and G. Ferguson. Actions and events in interval temporal logic. *Journal of Logic and Computation*, 4(5):531–579, October 1994.
3. S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite events for active databases: Semantics, contexts and detection. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *20th International Conference on Very Large Data Bases, September 12–15, 1994, Santiago, Chile proceedings*, pages 606–617, Los Altos, CA 94022, USA, 1994. Morgan Kaufmann Publishers.
4. A. Galton and J. C. Augusto. Two approaches to event definition. In R. Cicchetti, A. Hameurlain, and R. Traunmller, editors, *Proc. of Database and Expert Systems Applications 13th International Conference (DEXA'02)*, volume 2453 of *Lecture Notes in Computer Science*, pages 547–556, Aix-en-Provence, France, 2–6 September 2002. Springer-Verlag.
5. S. Gatziu and K.R. Dittrich. Events in an Active Object-Oriented Database System. In N.W. Paton and H.W. Williams, editors, *Proc. 1st Intl. Workshop on Rules in Database Systems (RIDS)*, Edinburgh, UK, September 1993. Springer-Verlag, Workshops in Computing.
6. N. Gehani, H. V. Jagadish, and O. Shmueli. COMPOSE: A system for composite specification and detection. In *Advanced Database Systems*, volume 759 of *Lecture Notes in Computer Science*. Springer, 1993.
7. A. Hinze and A. Voisard. A parameterized algebra for event notification services. In *Proceedings of the 9th International Symposium on Temporal Representation and Reasoning (TIME 2002), Manchester, UK*, 2002.
8. R. Kowalski and M. Sergot. A logic-based calculus of events. In J. W. Schmidt and C. Thanos, editors, *Foundations of Knowledge Base Management: Contributions from Logic, Databases, and Artificial Intelligence*, pages 23–55. Springer, Berlin, Heidelberg, 1989.
9. G. Liu, A. Mok, and P. Konana. A unified approach for specifying timing constraints and composite events in active real-time database systems. In *4th IEEE Real-Time Technology and Applications Symposium (RTAS '98)*, pages 199–209, Washington - Brussels - Tokyo, June 1998. IEEE.
10. J. Mellin and S. F. Adler. A formalized schema for event composition. In *Proc. 8th Int. Conf on Real-Time Computing Systems and Applications (RTCSA 2002)*, pages 201–210, Tokyo, Japan, 18–20 March 2002.
11. D. Zimmer and R. Unland. On the Semantics of Complex Events in Active Database Management Systems. In *Proceedings of the 15th International Conference on Data Engineering*, pages 392–399. IEEE Computer Society Press, 1999.