An event detection algebra for reactive systems

Jan Carlson and Björn Lisper
Department of Computer Science and Engineering
Mälardalen University, Sweden
`jan.carlson@mdh.se`   `bjorn.lisper@mdh.se`

Abstract

In reactive systems, execution is driven by external events to which the system should respond with appropriate actions. Such events can be simple, but systems are often supposed to react to sophisticated situations involving a number of simpler events occurring in accordance with some pattern. A systematic approach to handle this type of systems is to separate the mechanism for detecting composite events from the rest of the application logic.

In this paper, we present an event algebra for composite event detection. We show a number of algebraic laws that facilitate formal reasoning, and justify the algebra semantics by showing to what extent the operators comply with intuition. Finally, we present an implementation of the algebra, and identify a large subset of expressions for which detection can be performed with bounded resources.

1 Introduction

Many real-time and embedded systems are *reactive*, meaning that the execution is driven by external events to which the system should react with an appropriate response. For many applications, the system should react to complex event patterns, sometimes called composite events, rather than to a single event occurrence. A systematic approach to handle this type of systems is to separate the mechanism for detecting composite events from the rest of the application logic. The detection mechanism takes as input primitive events and detects occurrences of composite events which are used as input to the application logic. This separation of concerns facilitates design and analysis of reactive systems, as detection of complex events can be given a formal semantics independent from the application in which it is used, and the remaining application logic is free from auxiliary rules and information about partially completed patterns.

Example 1.1   Consider a system with input events including a button B, a pressure alarm P and a temperature alarm T, where one desired reaction is that the system should perform an action A when the button is pressed twice within two seconds, unless either of the alarms occurs in between. This can be achieved by a set of rules that specifies reactions to the three events, so that the combined behaviour implements the desired reaction. Alternatively, a separate detection mechanism can be used to define a composite event E that corresponds to the described situation, with a single rule stating that an occurrence of E should trigger the action A. The two approaches are illustrated by Figure 1.

The mechanism to detect composite events can be constructed as an event algebra, i.e., a number of operators from which expressions can be built that represent
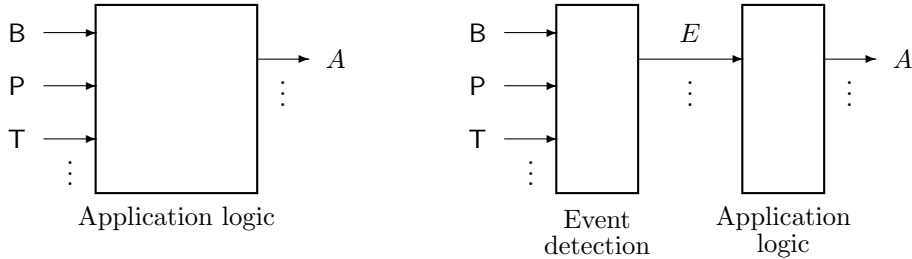
Figure 1: Implicit and explicit detection of composite events

the event patterns of interest. In this paper, we propose an event algebra that specifically targets applications with limited resources, such as embedded and real-time systems. We present a number of algebraic laws that facilitate formal reasoning, and justify the algebra semantics by showing to what extent the operators comply with intuition. Finally, we identify criteria under which detection can be performed with limited resources, and present a transformation algorithm that allows many expressions to be transformed into a form where these criteria are met.

The proposed algebra consists of five operators: The *disjunction* of A and B represents that either of A and B occurs, here denoted $A \lor B$. *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. A *sequence* $A;B$ is an occurrence of A followed by an occurrence of B. Finally, there is a *temporal restriction* A_τ which occurs when there is an occurrence of A shorter than τ time units.

Example 1.2 The composite event E from the previous example corresponds to the expression $(B;B)_2 - (P \lor T)$ in the algebra.

The operator semantics described informally above does not specify how to handle situations where an occurrence could participate in several occurrences of a composite event. For example, three occurences of A followed by two occurrences of B result in six occurrences of $A+B$. While this may be acceptable, or even desirable, in some applications, the memory requirements (each occurrence of A and B must be remembered forever) and the increasing number of simultaneous events means that it is unsuitable in many cases.

A common way to deal with this is to introduce variants of the operators that impose stronger constraints in addition to the basic conditions give above. For example, a possible sequence variant is to require that in addition to A occurring before B, this should be the most recent occurrence of A so far. Such variants can be defined by means of general restriction policies, where each combination of an operator and a restriction policy yields a operator variant with specific semantics. When restriction is applied to individual operator occurrences in the expression, as in existing event algebras with restriction policies, a user of the algebra must understand the interference from nested restrictions, and the effect of restriction on different operator combinations.

We have developed a novel restriction policy that is conceptually applied to the expression as a whole, rather than at the individual operators, which results in an algebra with simpler and more intuitive semantics. The policy is carefully designed so that applying it once at the top level is semantically consistent with applying it recursively to all subexpressions, which allows an efficient implementation.

As far as we know, previous research on event algebras has not addressed conformance to algebraic laws. In particular, event algebras suited for systems with

limited resources typically exhibit unintuitive semantics and poor algebraic properties.

The rest of this paper is organised as follows: Section 2 surveys related work. The algebra is defined in Section 3, followed by a presentation of the algebraic properties in Section 4. Section 5 presents an implementation, including an analysis of time and memory complexity. Section 6 describes a semantic-preserving transformation algorithm, and Section 7 concludes the paper.

2   Related work

The operators of our algebra, as well as the use of interval semantics and restricted detection, are influenced by work in the area of active databases. Snoop [5], Ode [8] and SAMOS [7] are examples of active database systems where an event algebra is used to specify the reactive behaviour. These 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. In the area of active databases, event algebras are often not given a formal semantics, and algebraic properties of the operators are not investigated. Also, resource efficiency is typically not a main concern.

Common to these systems is that they consider composite events to be instantaneous, i.e., an occurrence is associated with a single time instant, normally the time at which it can be detected. Galton and Augusto have shown that this results in unintended semantics for some operation compositions [6]. For example, an occurrence of A followed by B and then C, is accepted as an occurrence of the composite event $B;(A;C)$, since B occurs before the occurrence of $A;C$. They also present the core of an alternative, interval-based, semantics to handle these problems. We use a similar semantic base for our algebra, but we extend it with a restriction policy to allow the algebra to be implemented with limited resources while retaining the desired algebraic properties.

Liu et al. use Real Time Logic to define a system where composite events are expressed as timing constraints and handled by general timing constraint monitoring techniques. They present a mechanism for early detection of timing constraint violation, and show that upper bounds on memory and time can be derived [10].

In middleware platforms, event detection techniques are used to handle high volumes of event occurrences by allowing consumers to subscribe to certain event patterns rather than to single event types. Sánchez et al. present an event correlation language where event expressions are translated into nested Petri net like automata. [12].

Knowledge representation techniques use similar operators to reason about event occurrences. Rather than detecting complex events as they occur, they focus on how to express formally the fact that some event has occurred, and on defining inferences rules for this type of statements. Examples include Interval Calculus [1] and Event Calculus [9].

The event detection mechanisms described above provide no assistance to the developer in terms of algebraic properties or an event expression equivalence theory. In the cases where memory usage is addressed, for example by means of restriction policies, this results in complicated and non-intuitive semantics.

We propose an algebra for which a large class of composite events can be detected with limited resources. The algebra defined by a simple declarative semantics and we present a number of algebraic laws that facilitate formal reasoning, and supports the claim that the intuitive meaning of the operators is valid also for complex nested expressions. A preliminary version of the algebra, with less useful algebraic properties and with no memory bound, was described in a previous paper [3].

3 Declarative semantics

For simplicity, we assume a discrete time model throughout the paper. The declarative semantics of the algebra can be used with a dense time model as well, under restrictions that prevent primitive events that occur infinitely many times in a finite time interval.

Definition 3.1 *The temporal domain $\mathcal{T}$ is the set of all natural numbers.*

3.1 Primitive events

We assume that the system has a pre-defined set of primitive event types to which it should be able to react. 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.

For some primitive events, it is useful to associate additional information with each occurrence. For example, the occurrences of a temperature alarm might carry the measured temperature value, to be used in the responding action. These values are not manipulated by the algebra, only grouped and forwarded to the part of the system that reacts to the detected events.

Definition 3.2 *Let $\mathcal{P}$ be a finite set of identifiers that represent the primitive event types that are of interest to the system. For each identifier $p \in \mathcal{P}$, let $\mathrm{dom}(p)$ denote the domain from which the values of p are taken.*

Occurrences of primitive events are assumed to be instantaneous and atomic. In the algebra, they are represented by event instances that contain event type, occurrence time and a value. Formally, we represent a primitive instance as a singleton set, to allow primitive and complex instances to be treated uniformly.

Definition 3.3 *If $p \in \mathcal{P}$, $\upsilon \in \mathrm{dom}(p)$ and $\tau \in \mathcal{T}$, then the singleton set $\{\langle p, \upsilon, \tau \rangle\}$ is a primitive event instance.*

Together, the occurrences of a certain event type form an event stream. We allow simultaneous occurrences in general, but occurrences of the same primitive event type are assumed to be non-simultaneous.

Definition 3.4 *A primitive event stream is a set of primitive event instances all of which have the same identifier, and different times.*

The set of identifiers and the value domains capture static aspects of the system, and instances and event streams are dynamic concepts that describe what happens during a particular execution of the program. An interpretation is a formal representation of a single scenario, as it describes one particular case of primitive event occurrences.

Definition 3.5 *An interpretation is a function that maps each identifier $p \in \mathcal{P}$ to a primitive event stream containing instances with identifier p.*

Example 3.1 Let $\mathcal{P} = \{\mathsf{T}, \mathsf{P}\}$ with $\mathrm{dom}(\mathsf{T}) = \mathbb{N}$ and $\mathrm{dom}(\mathsf{P}) = \{\mathrm{high}, \mathrm{low}\}$. Now $S = \{\{\langle \mathsf{T}, 12, 2 \rangle\}, \{\langle \mathsf{T}, 14, 3 \rangle\}, \{\langle \mathsf{T}, 8, 5 \rangle\}\}$ and $S' = \{\{\langle \mathsf{P}, \mathrm{low}, 4 \rangle\}\}$ are examples of primitive event streams, and $\mathcal{I}$ such that $\mathcal{I}(\mathsf{T}) = S$ and $\mathcal{I}(\mathsf{P}) = S'$ is a possible interpretation.

3.2 Composite events

Composite events are represented by expressions built from the identifiers and the operators of the algebra.

Definition 3.6 *If $A \in \mathcal{P}$, then A is an event expression. If A and B are event expressions, and $\tau \in \mathcal{T}$, then $A \lor B$, $A+B$, $A-B$, $A;B$ and A_τ are event expressions.*

Next, we extend the concepts of instances and streams to composite events as well as primitive. The way in which instances are constructed is defined by the algebra semantics. For now, we only define their structure.

Definition 3.7 *An event instance is a union of n primitive event instances, where $0 < n$.*

Informally, an instance of a composite event represents the primitive event occurrences that caused an occurrence of the composite event. Since the semantics should be interval-based, we associate each instance with an interval, through the following definition.

Definition 3.8 *For an event instance a we define*

$$\begin{aligned}
\text{start}(a) &= \text{Min}_{\langle i,v,\tau \rangle \in a} \; (\tau) \\
\text{end}(a) \;\; &= \text{Max}_{\langle i,v,\tau \rangle \in a} \; (\tau)
\end{aligned}$$

The interval $[\text{start}(a), \text{end}(a)]$ can be thought of as the smallest interval which contains all the occurrences of primitive events that caused the occurrence of a. Note that a primitive event instance is an event instance, and if a is a primitive instance then $\text{start}(a) = \text{end}(a)$.

Example 3.2 Let $a = \{\langle \mathsf{T}, 12, 2 \rangle, \; \langle \mathsf{P}, \text{low}, 4 \rangle, \; \langle \mathsf{T}, 8, 5 \rangle\}$, then a is an event instance, and we have $\text{start}(a) = 2$ and $\text{end}(a) = 5$.

We also need a definition of general event streams. These will be used to represent all instances of a composite event. By this definition, a primitive event stream is an event stream, just as the names suggest.

Definition 3.9 *An event stream is a set of event instances.*

The naming convention is to use S, T and U for event streams, and A, B, C, etc. for event expressions. Lower case letters are used for event instances, and in general s belongs to the event stream S, etc.

3.3 Semantics

The interpretation provides the occurrences of the primitive events, by mapping each identifier to an event stream, and the role of the algebra semantics is to extend this mapping to composite events defined by event expressions.

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

Definition 3.10 *For event streams S and T, and $\tau \in \mathcal{T}$, define:*

$$\begin{aligned}
\text{dis}(S,T) \;\; &= S \cup T \\
\text{con}(S,T) &= \{s \cup t \mid s \in S \land t \in T\} \\
\text{neg}(S,T) &= \{s \mid s \in S \land \neg\exists t (t \in T \land \text{start}(s) \leq \text{start}(t) \land \text{end}(t) \leq \text{end}(s))\} \\
\text{seq}(S,T) \;\; &= \{s \cup t \mid s \in S \land t \in T \land \text{end}(s) < \text{start}(t)\} \\
\text{tim}(S,\tau) \;\; &= \{s \mid s \in S \land \text{end}(s) - \text{start}(s) \leq \tau\}
\end{aligned}$$

The semantics of the algebra is defined by recursively applying the corresponding function for each operation in the expression.

Definition 3.11 *The meaning of an event expression for a given interpretation $\mathcal{I}$ is defined as follows:*

$$
\begin{array}{llllll}
[\![A]\!]^{\mathcal{I}} & = & \mathcal{I}(A) \text{ if } A \in \mathcal{P} & \quad [\![A\!-\!B]\!]^{\mathcal{I}} & = & \mathrm{neg}([\![A]\!]^{\mathcal{I}}, [\![B]\!]^{\mathcal{I}}) \\
[\![A \vee B]\!]^{\mathcal{I}} & = & \mathrm{dis}([\![A]\!]^{\mathcal{I}}, [\![B]\!]^{\mathcal{I}}) & \quad [\![A;B]\!]^{\mathcal{I}} & = & \mathrm{seq}([\![A]\!]^{\mathcal{I}}, [\![B]\!]^{\mathcal{I}}) \\
[\![A+B]\!]^{\mathcal{I}} & = & \mathrm{con}([\![A]\!]^{\mathcal{I}}, [\![B]\!]^{\mathcal{I}}) & \quad [\![A_\tau]\!]^{\mathcal{I}} & = & \mathrm{tim}([\![A]\!]^{\mathcal{I}}, \tau)
\end{array}
$$

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

These definitions result in an algebra with simple semantics and intuitive algebraic properties, but it can not be implemented with limited resources. To deal with resource limitations, we define a formal restriction policy, and require only that an implementation should compute a valid restriction of the event stream specified by the algebra semantics.

Formally, the restriction policy is defined as a relation *rem*, where $rem(S, S')$ means that S' is a valid restriction of S. Alternatively, it can be seen as a non-deterministic restriction function, or a family of acceptable restriction functions. For reasons of repeatability, it is typically desirable that an implementation of the algebra is deterministic. From a theoretical point of view, however, we prefer to leave as many detailed design decisions as possible open, as we can ensure that any implementation which is consistent with the restriction policy relation is guaranteed to have the properties described in the paper.

The basis of the restriction policy it that the restricted event stream should not contain multiple instances with the same end time, as this is one of the efficiency issues. Informally, from the instances with the same end time, the restriction policy keeps exactly one with maximal start time.

Definition 3.12 *For two event streams, S and S', $rem(S, S')$ holds if the following conditions hold:*

1. $S' \subseteq S$

2. $\forall s \ (s \in S \ \Rightarrow \ \exists s'(s' \in S' \wedge \mathrm{start}(s) \leq \mathrm{start}(s') \wedge \mathrm{end}(s) = \mathrm{end}(s')))$

3. $\forall s, s' \ ((s \in S' \wedge s' \in S' \wedge \mathrm{end}(s) = \mathrm{end}(s')) \ \Rightarrow \ s = s')$

Rather than computing $[\![A]\!]$ for a given event expression A, an implementation of the algebra computes an event stream S' for which $rem([\![A]\!], S')$ holds. For the user of the algebra, this means that at any time when there is one or more occurrences of A, one of them will be detected.

4   Properties

To aid a user of this algebra, we present a selection of algebraic laws. These laws facilitate reasoning, both formally and informally, about the algebra and any system in which it is embedded. They also show to what extent the operators behave according to intuition. For this, we first define expression equivalence.

Definition 4.1 *For event expressions A and B we define $A \equiv B$ to hold if $[\![A]\!]^{\mathcal{I}} = [\![B]\!]^{\mathcal{I}}$ for any interpretation $\mathcal{I}$.*

Trivially, $\equiv$ is an equivalence relation. Moreover, the following theorem shows that it satisfies the substitutive condition, and hence defines structural congruence over event expressions.

Theorem 4.1 *If $A \equiv A'$, $B \equiv B'$ and $\tau \in \mathcal{T}$, then we have $A \vee B \equiv A' \vee B'$, $A + B \equiv A' + B'$, $A;B \equiv A';B'$, $A - B \equiv A' - B'$ and $A_\tau \equiv A'_\tau$.*

Proof: This follows trivially from definitions 3.10 and 4.1. $\qquad\qquad\square$

The following laws describe the properties of the disjunction, conjunction and sequence operators, and how they distribute.

Theorem 4.2 *For event expressions A, B and C, the following laws hold.*

$$
\begin{array}{rrcl}
1. & A \vee A & \equiv & A \\
2. & A \vee B & \equiv & B \vee A \\
3. & A + B & \equiv & B + A \\
4. & A \vee (B \vee C) & \equiv & (A \vee B) \vee C \\
5. & A + (B + C) & \equiv & (A + B) + C \\
6. & A;(B;C) & \equiv & (A;B);C \\
7. & (A \vee B) + C & \equiv & (A + C) \vee (B + C) \\
8. & (A \vee B);C & \equiv & (A;C) \vee (B;C) \\
9. & A;(B \vee C) & \equiv & (A;B) \vee (A;C)
\end{array}
$$

Corollary 4.1

$$
\begin{array}{rrcl}
10. & A + (B \vee C) & \equiv & (A + B) \vee (A + C)
\end{array}
$$

Proof: Most of the laws follow trivially from definitions 4.1, 3.10 and 3.11.

1. $[\![A \vee A]\!] = \mathrm{dis}([\![A]\!], [\![A]\!]) = [\![A]\!] \cup [\![A]\!] = [\![A]\!]$

2. $[\![A \vee B]\!] = \mathrm{dis}([\![A]\!], [\![B]\!]) = \mathrm{dis}([\![B]\!], [\![A]\!]) = [\![B \vee A]\!]$

3. $[\![A + B]\!] = \mathrm{con}([\![A]\!], [\![B]\!]) = \mathrm{con}([\![B]\!], [\![A]\!]) = [\![B + A]\!]$

4. $[\![A \vee (B \vee C)]\!] = [\![A]\!] \cup [\![B]\!] \cup [\![C]\!] = [\![(A \vee B) \vee C]\!]$

5. $[\![A + (B + C)]\!] = \mathrm{con}([\![A]\!], \mathrm{con}([\![B]\!], [\![C]\!])) =$
   $\{a \cup b \cup c \mid a \in [\![A]\!] \wedge b \in [\![B]\!] \wedge c \in [\![C]\!]) = [\![(A + B) + C]\!]$

6. $[\![A;(B;C)]\!] = \{a \cup e \mid a \in [\![A]\!] \wedge e \in \{b \cup c \mid b \in [\![B]\!] \wedge c \in [\![C]\!] \wedge$
   $\mathrm{end}(b) < \mathrm{start}(c)\} \wedge \mathrm{end}(a) < \mathrm{start}(e)\} =$
   $\{a \cup b \cup c \mid a \in [\![A]\!] \wedge b \in [\![B]\!] \wedge c \in [\![C]\!] \wedge \mathrm{end}(a) < \mathrm{start}(b) \wedge \mathrm{end}(b) < \mathrm{start}(c)\} =$
   $[\![(A;B);C]\!]$

7. $[\![(A \vee B) + C]\!] = \mathrm{con}(\mathrm{dis}([\![A]\!], [\![B]\!]), [\![C]\!]) = \mathrm{con}(([\![A]\!] \cup [\![B]\!]), [\![C]\!]) =$
   $\{e \cup c \mid e \in [\![A]\!] \cup [\![B]\!] \wedge c \in [\![C]\!]\} =$
   $\{a \cup c \mid a \in [\![A]\!] \wedge c \in [\![C]\!]\} \cup \{b \cup c \mid b \in [\![A]\!] \wedge c \in [\![C]\!]\} =$
   $\mathrm{con}([\![A]\!], [\![C]\!]) \cup \mathrm{con}([\![B]\!], [\![C]\!]) = [\![(A + C) \vee (B + C)]\!]$

8. $[\![(A \vee B);C]\!] = \{e \cup c \mid e \in [\![A]\!] \cup [\![B]\!] \wedge c \in [\![C]\!] \wedge \mathrm{end}(e) < \mathrm{start}(c)\} =$
   $\{a \cup c \mid a \in [\![A]\!] \wedge c \in [\![C]\!] \wedge \mathrm{end}(a) < \mathrm{start}(c)\} \cup$
   $\{b \cup c \mid b \in [\![B]\!] \wedge c \in [\![C]\!] \wedge \mathrm{end}(b) < \mathrm{start}(c)\} =$
   $[\![(A;C) \vee (B;C)]\!]$

9. $[\![A;(B \vee C)]\!] = \{a \cup e \mid a \in [\![A]\!] \wedge e \in [\![B]\!] \cup [\![C]\!] \wedge \mathrm{end}(a) < \mathrm{start}(e)\} =$
   $\{a \cup b \mid a \in [\![A]\!] \wedge b \in [\![B]\!] \wedge \mathrm{end}(a) < \mathrm{start}(b)\} \cup$
   $\{a \cup c \mid a \in [\![A]\!] \wedge c \in [\![C]\!] \wedge \mathrm{end}(a) < \mathrm{start}(c)\} =$
   $[\![(A;B) \vee (A;C)]\!]$

10. This follows from laws 2, 3 and 7.

$\square$

Next, we present a set of laws for negation. To simplify the proofs, we introduce the following predicate.

Definition 4.2 *For an event stream S, and $\tau, \tau' \in \mathcal{T}$, define $empty(S, \tau, \tau')$ to hold if $\neg \exists s (s \in S \wedge \tau \leq \text{start}(s) \wedge \text{end}(s) \leq \tau')$*

Trivially, $a \in [\![A - B]\!]$ iff $a \in [\![A]\!]$ and $empty([\![B]\!], \text{start}(a), \text{end}(a))$.

Theorem 4.3 *For event expressions A, B and C, the following laws hold.*

$$
\begin{array}{rlcl}
11. & (A-B)-C & \equiv & A-(B \vee C) \\
12. & A-(B-B) & \equiv & A \\
13. & (A \vee B)-C & \equiv & (A-C) \vee (B-C) \\
14. & (A+B)-C & \equiv & ((A-C)+B)-C \\
15. & (A;B)-C & \equiv & ((A-C);B)-C \\
16. & (A;B)-C & \equiv & (A;(B-C))-C
\end{array}
$$

Corollary 4.2

$$
\begin{array}{rlcl}
17. & (A-B)-B & \equiv & A-B \\
18 & (A-B)-C & \equiv & (A-C)-B \\
19. & (A \vee B)-C & \equiv & ((A-C) \vee B)-C \\
20. & (A \vee B)-C & \equiv & (A \vee (B-C))-C \\
21. & (A+B)-C & \equiv & (A+(B-C))-C \\
22. & (A-B)-C & \equiv & ((A-C)-B)-C
\end{array}
$$

Proof: Here, $\equiv^{23}$ denotes that the equivalence follows from law number 23, etc.

11. $a \in [\![(A-B)-C]\!] \Leftrightarrow$
    $a \in [\![A-B]\!] \wedge empty([\![C]\!], \text{start}(a), \text{end}(a)) \Leftrightarrow$
    $a \in [\![A]\!] \wedge empty([\![B]\!], \text{start}(a), \text{end}(a)) \wedge$
    $empty([\![C]\!], \text{start}(a), \text{end}(a)) \Leftrightarrow$
    $a \in [\![A]\!] \wedge empty([\![B]\!] \cup [\![C]\!], \text{start}(a), \text{end}(a)) \Leftrightarrow$
    $a \in [\![A-(B \vee C)]\!]$

12. Since $[\![B-B]\!]^{\mathcal{I}} = \emptyset$ for any interpretation $\mathcal{I}$, it follows that $[\![A-(B-B)]\!] = [\![A]\!]$.

13. $[\![(A \vee B)-C]\!] =$
    $\{e \mid e \in [\![A]\!] \cup [\![B]\!] \wedge empty([\![C]\!], \text{start}(e), \text{end}(e))\} =$
    $\{a \mid a \in [\![A]\!] \wedge empty([\![C]\!], \text{start}(a), \text{end}(a))\} \cup$
    $\{b \mid b \in [\![B]\!] \wedge empty([\![C]\!], \text{start}(b), \text{end}(b))\} =$
    $[\![(A-C)]\!] \cup [\![(B-C)]\!] =$
    $[\![(A-C) \vee (B-C)]\!]$

14. $e \in [\![((A-C)+B)-C]\!] \Leftrightarrow$
    $e = a \cup b \wedge a \in [\![(A-C)]\!] \wedge b \in [\![B]\!] \wedge empty([\![C]\!], \text{start}(e), \text{end}(e)) \Leftrightarrow$
    $e = a \cup b \wedge a \in [\![A]\!] \wedge b \in [\![B]\!] \wedge empty([\![C]\!], \text{start}(e), \text{end}(e)) \wedge$
    $empty([\![C]\!], \text{start}(a), \text{end}(a))$

    Since $\text{start}(e) \leq \text{start}(a)$ and $\text{end}(a) \leq \text{end}(e)$, this is equivalent to
    $e = a \cup b \wedge a \in [\![A]\!] \wedge b \in [\![B]\!] \wedge empty([\![C]\!], \text{start}(e), \text{end}(e)) \Leftrightarrow$
    $e = [\![(A+B)-C]\!]$

15. $e \in [\![(A;B)-C]\!] \Leftrightarrow$
    $e = a \cup b \wedge \text{end}(a) < \text{start}(b) \wedge a \in [\![A]\!] \wedge b \in [\![B]\!] \wedge$
    $empty([\![C]\!], \text{start}(a), \text{end}(b)) \Leftrightarrow$
    $e = a \cup b \wedge \text{end}(a) < \text{start}(b) \wedge a \in [\![A]\!] \wedge b \in [\![B]\!] \wedge$

8

$empty(\llbracket C \rrbracket, \mathrm{start}(a), \mathrm{end}(b)) \wedge empty(\llbracket C \rrbracket, \mathrm{start}(a), \mathrm{end}(a)) \Leftrightarrow$
$e = a \cup b \wedge \mathrm{end}(a) < \mathrm{start}(b) \wedge a \in \llbracket A - C \rrbracket \wedge b \in \llbracket B \rrbracket \wedge$
$empty(\llbracket C \rrbracket, \mathrm{start}(a), \mathrm{end}(b)) \Leftrightarrow$
$e \in \llbracket (A - C); B \rrbracket \wedge empty(\llbracket C \rrbracket, \mathrm{start}(e), \mathrm{end}(e)) \Leftrightarrow$
$e \in \llbracket ((A - C); B) - C \rrbracket$

16. $e \in \llbracket (A;B) - C \rrbracket \Leftrightarrow$
$e = a \cup b \wedge \mathrm{end}(a) < \mathrm{start}(b) \wedge a \in \llbracket A \rrbracket \wedge b \in \llbracket B \rrbracket \wedge$
$empty(\llbracket C \rrbracket, \mathrm{start}(a), \mathrm{end}(b)) \Leftrightarrow$
$e = a \cup b \wedge \mathrm{end}(a) < \mathrm{start}(b) \wedge a \in \llbracket A \rrbracket \wedge b \in \llbracket B \rrbracket \wedge$
$empty(\llbracket C \rrbracket, \mathrm{start}(a), \mathrm{end}(b)) \wedge empty(\llbracket C \rrbracket, \mathrm{start}(b), \mathrm{end}(b)) \Leftrightarrow$
$e = a \cup b \wedge \mathrm{end}(a) < \mathrm{start}(b) \wedge a \in \llbracket A \rrbracket \wedge b \in \llbracket B - C \rrbracket \wedge$
$empty(\llbracket C \rrbracket, \mathrm{start}(a), \mathrm{end}(b)) \Leftrightarrow$
$e \in \llbracket A; (B - C) \rrbracket \wedge empty(\llbracket C \rrbracket, \mathrm{start}(e), \mathrm{end}(e)) \Leftrightarrow$
$e \in \llbracket (A; (B - C)) - C \rrbracket$

17. This follows from laws 1 and 13.

18. This follows from laws 2 and 11.

19. $((A - C) \vee B) - C \equiv^{13} ((A - C) - C) \vee (B - C) \equiv^{17} (A - C) \vee (B - C) \equiv^{13} (A \vee B) - C$

20. This follows from laws 2 and 13.

21. This follows from laws 3 and 14.

22. $((A - C) - B) - C \equiv^{18} ((A - B) - C) - C \equiv^{17} (A - B) - C$

$\square$

The following laws describe how temporal restrictions can be propagated through an expression. In Section 6, these laws are used to define an algorithm for transforming event expressions into an equivalent expressions that can be detected more efficiently.

Theorem 4.4 *For event expressions A, B and C, and $\tau \in \mathcal{T}$, the following laws hold.*

$$
\begin{array}{rrcll}
23. & A & \equiv & A_\tau & \textit{if } A \in \mathcal{P} \\
24. & (A_\tau)_{\tau'} & \equiv & A_{\min(\tau, \tau')} & \\
25. & (A \vee B)_\tau & \equiv & A_\tau \vee B_\tau & \\
26. & (A + B)_\tau & \equiv & (A_\tau + B)_\tau & \\
27. & (A - B)_\tau & \equiv & A_\tau - B & \\
28. & (A - B)_\tau & \equiv & (A - B_\tau)_\tau & \\
29. & (A;B)_\tau & \equiv & (A_\tau;B)_\tau & \\
30. & (A;B)_\tau & \equiv & (A;B_\tau)_\tau &
\end{array}
$$

Corollary 4.3

$$
\begin{array}{rrcl}
31. & (A_\tau)_{\tau'} & \equiv & (A_{\tau'})_\tau \\
32. & (A \vee B)_\tau & \equiv & (A_\tau \vee B)_\tau \\
33. & (A \vee B)_\tau & \equiv & (A \vee B_\tau)_\tau \\
34. & A_\tau \vee B_{\tau'} & \equiv & (A_\tau \vee B_{\tau'})_{\max(\tau, \tau')} \\
35. & (A + B)_\tau & \equiv & (A + B_\tau)_\tau \\
36. & (A - B)_\tau & \equiv & A_\tau - B_\tau
\end{array}
$$

Proof:

23. $A \in \mathcal{P}$ implies that $\mathrm{end}(a) - \mathrm{start}(a) = 0$ for any $a \in \llbracket A \rrbracket$, which means that $\llbracket A \rrbracket = \llbracket A_\tau \rrbracket$.

9

24. $\llbracket (A_\tau)_{\tau'} \rrbracket = \{a \mid a \in \llbracket A \rrbracket \wedge \text{end}(a) - \text{start}(a) \leq \tau \wedge \text{end}(a) - \text{start}(a) \leq \tau'\} =$
$\{a \mid a \in \llbracket A \rrbracket \wedge \text{end}(a) - \text{start}(a) \leq \min(\tau, \tau')\} = \llbracket A_{\min(\tau,\tau')} \rrbracket$

25. $\llbracket (A \vee B)_\tau \rrbracket = \{e \mid e \in A \cup B \wedge \text{end}(e) - \text{start}(e) \leq \tau\} =$
$\{a \mid a \in A \wedge \text{end}(a) - \text{start}(a) \leq \tau\} \cup \{b \mid b \in B \wedge \text{end}(b) - \text{start}(b) \leq \tau\} =$
$\llbracket A_\tau \rrbracket \cup \llbracket B_\tau \rrbracket = \llbracket A_\tau \vee B_\tau \rrbracket$

26. $e \in \llbracket (A_\tau + B)_\tau \rrbracket \Leftrightarrow e \in \llbracket A_\tau + B \rrbracket \wedge \text{end}(e) - \text{start}(e) \leq \tau \Leftrightarrow$
$e = a \cup b \wedge a \in \llbracket A_\tau \rrbracket \wedge b \in \llbracket B \rrbracket \wedge \text{end}(e) - \text{start}(e) \leq \tau \Leftrightarrow$
$e = a \cup b \wedge a \in \llbracket A \rrbracket \wedge \text{end}(a) - \text{start}(a) \leq \tau \wedge b \in \llbracket B \rrbracket \wedge \text{end}(e) - \text{start}(e) \leq \tau$

    Since $\text{end}(a) \leq \text{end}(e)$ and $\text{start}(e) \leq \text{start}(a)$, we have $\text{end}(a) - \text{start}(a) \leq \text{end}(e) - \text{start}(e)$ so $\text{end}(e) - \text{start}(e) \leq \tau \Rightarrow \text{end}(a) - \text{start}(a) \leq \tau$. Thus, the last formula above is equivalent to
$e = a \cup b \wedge a = \llbracket A \rrbracket \wedge b = \llbracket B \rrbracket \wedge \text{end}(e) - \text{start}(e) \leq \tau \Leftrightarrow$
$e \in \llbracket A + B \rrbracket \wedge \text{end}(e) - \text{start}(e) \leq \tau \Leftrightarrow e \in \llbracket (A + B)_\tau \rrbracket$.

27. $\llbracket (A - B)_\tau \rrbracket = \{a \mid a \in \llbracket A - B \rrbracket \wedge \text{end}(a) - \text{start}(a) \leq \tau\} =$
$\{a \mid a \in \llbracket A \rrbracket \wedge \text{end}(a) - \text{start}(a) \leq \tau \wedge empty(\llbracket B \rrbracket, \text{start}(a), \text{end}(a))\} =$
$\{a \mid a \in \llbracket A_\tau \rrbracket \wedge empty(\llbracket B \rrbracket, \text{start}(a), \text{end}(a))\} =$
$\llbracket A_\tau - B \rrbracket$

28. $\llbracket (A - B_\tau)_\tau \rrbracket = \{a \mid a \in \llbracket A \rrbracket \wedge \text{end}(a) - \text{start}(a) \leq \tau \wedge empty(\llbracket B_\tau \rrbracket, \text{start}(a), \text{end}(a))\} =$

    $\{a \mid a \in \llbracket A \rrbracket \wedge \text{end}(a) - \text{start}(a) \leq \tau \wedge \neg \exists b (b \in \llbracket B \rrbracket \wedge \text{start}(a) \leq \text{start}(b) \wedge \text{end}(b) \leq \text{end}(a) \wedge \text{end}(b) - \text{start}(b) \leq \tau)\}$

    Since $\text{end}(a) - \text{start}(a) \leq \tau$, $\text{start}(a) \leq \text{start}(b)$ and $\text{end}(b) \leq \text{end}(a)$ implies $\text{end}(b) - \text{start}(b) \leq \tau$, that constraint can be removed without changing the resulting set. Thus, the set above is equivalent to
$\{a \mid a \in \llbracket A \rrbracket \wedge \text{end}(a) - \text{start}(a) \leq \tau \wedge empty(\llbracket B \rrbracket, \text{start}(a), \text{end}(a))\} =$
$\llbracket (A - B)_\tau \rrbracket$.

29. $\llbracket (A; B_\tau)_\tau \rrbracket =$
$\{a \cup b \mid a \in \llbracket A \rrbracket \wedge b \in \llbracket B_\tau \rrbracket \wedge \text{end}(a) < \text{start}(b) \wedge \text{end}(b) - \text{start}(a) \leq \tau\} =$
$\{a \cup b \mid a \in \llbracket A \rrbracket \wedge b \in \llbracket B \rrbracket \wedge \text{end}(b) - \text{start}(b) \leq \tau \wedge \text{end}(a) < \text{start}(b) \wedge$
$\text{end}(b) - \text{start}(a) \leq \tau\}$

    Since $\text{end}(a) < \text{start}(b)$ and $\text{end}(b) - \text{start}(a) \leq \tau$ implies $\text{end}(b) - \text{start}(b) \leq \tau$, this criteria can be dropped without changing the meaning. Thus, the set above is equivalent to
$\{a \cup b \mid a \in \llbracket A \rrbracket \wedge b \in \llbracket B \rrbracket \wedge \text{end}(a) < \text{start}(b) \wedge \text{end}(b) - \text{start}(a) \leq \tau\} = \llbracket (A; B)_\tau \rrbracket$

30. $\llbracket (A_\tau; B)_\tau \rrbracket =$
$\{a \cup b \mid a \in \llbracket A_\tau \rrbracket \wedge b \in \llbracket B \rrbracket \wedge \text{end}(a) < \text{start}(b) \wedge \text{end}(b) - \text{start}(a) \leq \tau\} =$
$\{a \cup b \mid a \in \llbracket A \rrbracket \wedge \text{end}(a) - \text{start}(a) \leq \tau \wedge b \in \llbracket B \rrbracket \wedge \text{end}(a) < \text{start}(b) \wedge$
$\text{end}(b) - \text{start}(a) \leq \tau\}$

    Since $\text{end}(a) < \text{start}(b)$ and $\text{end}(b) - \text{start}(a) \leq \tau$ implies $\text{end}(a) - \text{start}(a) \leq \tau$, this criteria can be dropped without changing the meaning. Thus, the set above is equivalent to
$\{a \cup b \mid a \in \llbracket A \rrbracket \wedge b \in \llbracket B \rrbracket \wedge \text{end}(a) < \text{start}(b) \wedge \text{end}(b) - \text{start}(a) \leq \tau\} = \llbracket (A; B)_\tau \rrbracket$

31. $(A_\tau)_{\tau'} \equiv^{28} A_{\min(\tau,\tau')} \equiv A_{\min(\tau,\tau')} \equiv^{28} (A_{\tau'})_\tau$

32. $(A \vee B)_\tau \equiv^{25} A_\tau \vee B_\tau \equiv^{24} (A_\tau)_\tau \vee B_\tau \equiv^{25} (A \vee B)_\tau$

33. This follows trivially from laws 2 and 32.

34. $(A_\tau \vee B_{\tau'})_{\max(\tau,\tau')} \equiv^{25} (A_\tau)_{\max(\tau,\tau')} \vee (B_{\tau'})_{\max(\tau,\tau')} \equiv^{24}$
    $A_{\min(\tau,\max(\tau,\tau'))} \vee B_{\min(\tau',\max(\tau,\tau'))} \equiv A_\tau \vee B_{\tau'}$

35. This follows trivially from laws 3 and 26.

36. $(A - B)_\tau \equiv^{28} (A - B_\tau)_\tau \equiv^{27} A_\tau - B_\tau$

$\square$

The laws identify expressions that are semantically equivalent, but in order to handle resource limitations, we expect an implementation of the algebra to compute an event stream S' such that $rem(\llbracket A \rrbracket, S')$. As a result, detecting A might yield a different stream than detecting A', even when $A \equiv A'$. Consequently, it should be clearified to what extent the laws presented above are still applicable when restriction is applied.

Trivially, if $A \equiv A'$ then a valid restriction to $\llbracket A \rrbracket$ is also a valid restriction to $\llbracket A' \rrbracket$. However, while the restriction policy is defined as a relation, an implementation produces a single restricted stream. As a result, detecting A might yield a different stream than detecting A'. Although not identical, we will show that the two streams are closely related.

Definition 4.3 *For event streams S and T, define $S \cong T$ to hold if the following holds:* $\{\langle \mathrm{start}(s), \mathrm{end}(s) \rangle \mid s \in S\} = \{\langle \mathrm{start}(t), \mathrm{end}(t) \rangle \mid t \in T\}$

Trivially, $\cong$ is an equivalence relation.

Theorem 4.5 *If $rem(S,T)$ and $rem(S,T')$ holds, then $T \cong T'$*

Proof: Take any $t \in T$. Then, since $T \subseteq S$, $t \in S$. By the second condition in the definition of rem, there is some $t' \in T'$ such that $\mathrm{start}(t) < \mathrm{start}(t')$ and $\mathrm{end}(t) = \mathrm{end}(t')$. We also have $t' \in S$, and thus there is some $t'' \in T$ such that $\mathrm{start}(t') \leq \mathrm{start}(t'')$ and $\mathrm{end}(t') = \mathrm{end}(t'')$. According to the third condition in the definition of rem this implies $t = t''$, which means that we have $\mathrm{start}(t) \leq \mathrm{start}(t') \leq \mathrm{start}(t)$ and thus $\mathrm{start}(t') = \mathrm{start}(t)$. So, for any $t \in T$ there is a $t' \in T'$ with the same start and end time. Trivially, the opposite holds as well. $\square$

Corollary 4.4 *If $A \equiv A'$, $rem(\llbracket A \rrbracket, T)$ and $rem(\llbracket A \rrbracket, T')$, then $T \cong T'$.*

Thus, $A \equiv A'$ ensures that for any implementation consistent with the restriction policy, the instances found when detecting A and A' have the same start and end times. Of course, the detected instances are also guaranteed to belong to the event stream defined by the algebra semantics (which according to $A \equiv A'$ are the same).

In order to get the desired efficiency, all subexpressions of an expression must be detected in an efficient way, and thus the restriction policy must be applied recursively to every subexpression. This scenario would normally require a user of the algebra to understand how the restrictions in different subexpressions interfer with each other, and how they effect different operator combinations. To avoid this, the operators and the restriction policy have been carefully designed to support the following theorem. Informally, it states that restricting the subexpressions as well as the whole expression gives a result which is valid also for the case when restriction is applied only at the top level.

Theorem 4.6 *If $rem(S, S')$ and $rem(T, T')$ holds, than for any event stream U and $\tau \in \mathcal{T}$ the following implications hold:*

$$
\begin{array}{lrcl}
1. & rem(\mathrm{dis}(S', T'),\ U) & \Rightarrow & rem(\mathrm{dis}(S, T),\ U) \\
2. & rem(\mathrm{con}(S', T'),\ U) & \Rightarrow & rem(\mathrm{con}(S, T),\ U) \\
3. & rem(\mathrm{neg}(S', T'),\ U) & \Rightarrow & rem(\mathrm{neg}(S, T),\ U) \\
4. & rem(\mathrm{seq}(S', T'),\ U) & \Rightarrow & rem(\mathrm{seq}(S, T),\ U) \\
5. & rem(\mathrm{tim}(S', \tau),\ U) & \Rightarrow & rem(\mathrm{tim}(S, \tau),\ U)
\end{array}
$$

Proof:

1. Assume $rem(\mathrm{dis}(S',T'),U)$. For any $u \in U$ we have $u \in \mathrm{dis}(S',T')$ and thus $u \in S' \cup T'$. Then, since $S' \subseteq S$ and $T' \subseteq T$, we have $u \in S \cup T$, implying $u \in \mathrm{dis}(S,T)$. Thus $U \subseteq \mathrm{dis}(S,T)$, which satisfies the first constraint in the definition of rem.

   Next, take an arbitrary $u \in \mathrm{dis}(S,T)$. Then $u \in S \cup T$ and by the definition of rem there exists an $u' \in S' \cup T'$ with $\mathrm{start}(u) \leq \mathrm{start}(u')$ and $\mathrm{end}(u') = \mathrm{end}(u)$. We have $u' \in \mathrm{dis}(S',T')$ and thus $rem(\mathrm{dis}(S',T'),U)$ implies that there exists an $u'' \in U$ such that $\mathrm{start}(u') \leq \mathrm{start}(u'')$ and $\mathrm{end}(u'') = \mathrm{end}(u')$. Since this means that $\mathrm{start}(u) \leq \mathrm{start}(u'')$ and $\mathrm{end}(u'') = \mathrm{end}(u)$, the second constraint in the definition of rem is satisfied.

   Finally, $rem(\mathrm{dis}(S',T'),U)$ ensures that all instances in U have different end times. Together, this gives $rem(\mathrm{dis}(S,T),U)$.

2. Assume $rem(\mathrm{con}(S',T'),U)$. For any $u \in U$ we have $u \in \mathrm{con}(S',T')$ and thus $u = s \cup t$ with $s \in S'$ and $t \in T'$. By the subset requirement in the definition of rem, $s \in S$ and $t \in T$. So $u \in \mathrm{con}(S,T)$ and thus $U \subseteq \mathrm{con}(S,T)$.

   Next, take an arbitrary $u \in \mathrm{con}(S,T)$. Then $u = s \cup t$ with $s \in S$ and $t \in T$, and by the definition of rem there exists $s' \in S'$ and $t' \in T'$ with $\mathrm{start}(s) \leq \mathrm{start}(s')$, $\mathrm{end}(s') = \mathrm{end}(s)$, $\mathrm{start}(t) \leq \mathrm{start}(t')$ and $\mathrm{end}(t') = \mathrm{end}(t)$. Let $u' = s' \cup t'$. Now $u' \in \mathrm{con}(S',T')$ with $\mathrm{start}(u) \leq \mathrm{start}(u')$ and $\mathrm{end}(u') = \mathrm{end}(u)$. This means that there exists some $u'' \in U$ with $\mathrm{start}(u) \leq \mathrm{start}(u'')$ and $\mathrm{end}(u'') = \mathrm{end}(u)$, which satisfies the second constraint in the definition of rem.

   Finally, $rem(\mathrm{con}(S',T'),U)$ ensures that all instances in U have different end times. Together, this gives $rem(\mathrm{con}(S,T),U)$.

3. Assume $rem(\mathrm{neg}(S',T'),U)$. For any $u \in U$ we have $u \in \mathrm{neg}(S',T')$ and thus $u \in S'$. By the subset requirement in the definition of rem, $u \in S$. If there exists a $t \in T$ with $\mathrm{start}(u) \leq \mathrm{start}(t)$ and $\mathrm{end}(t) \leq \mathrm{end}(u)$, then there must exist some $t' \in T'$ with $\mathrm{start}(t) \leq \mathrm{start}(t')$ and $\mathrm{end}(t') = \mathrm{end}(t)$ which contradicts the fact that $u \in \mathrm{neg}(S',T')$. Since no such t can exist, we have $u \in \mathrm{neg}(S,T)$ and thus $U \subseteq \mathrm{neg}(S,T)$.

   Next, take an arbitrary $u \in \mathrm{neg}(S,T)$. Then $u \in S$ and there exists an $u' \in S'$ with $\mathrm{start}(u) \leq \mathrm{start}(u')$, $\mathrm{end}(u') = \mathrm{end}(u)$. If there exists a $t \in T'$ with $\mathrm{start}(u') \leq \mathrm{start}(t)$ and $\mathrm{end}(t) \leq \mathrm{end}(u')$, then the fact that $t \in T$ contradicts $u \in \mathrm{neg}(S,T)$. Since no such t can exist, we have that $u' \in \mathrm{neg}(S',T')$. This means that there exists some $u'' \in U$ with $\mathrm{start}(u') \leq \mathrm{start}(u'')$ and $\mathrm{end}(u'') = \mathrm{end}(u')$, and thus $\mathrm{start}(u) \leq \mathrm{start}(u'')$ and $\mathrm{end}(u'') = \mathrm{end}(u)$, which satisfies the second constraint in the definition of rem.

   Finally, $rem(\mathrm{neg}(S',T'),U)$ ensures that all instances in U have different end times. Together, this gives $rem(\mathrm{neg}(S,T),U)$.

4. Assume $rem(\mathrm{seq}(S',T'),U)$. For any $u \in U$ we have $u \in \mathrm{seq}(S',T')$ and thus $u = s \cup t$ with $s \in S'$, $t \in T'$ and $\mathrm{end}(s) < \mathrm{start}(t)$. By the subset requirement in the definition of rem, $s \in S$ and $t \in T$. So $u \in \mathrm{seq}(S,T)$ and thus $U \subseteq \mathrm{seq}(S,T)$.

   Next, take an arbitrary $u \in \mathrm{seq}(S,T)$. Then $u = s \cup t$ with $s \in S$, $t \in T$ and $\mathrm{end}(s) < \mathrm{start}(t)$. By the definition of rem there exists $s' \in S'$ and $t \cup T'$ with $\mathrm{start}(s) \leq \mathrm{start}(s')$, $\mathrm{end}(s') = \mathrm{end}(s)$, $\mathrm{start}(t) \leq \mathrm{start}(t')$ and $\mathrm{end}(t') = \mathrm{end}(t)$. Let $u' = s' \cup t'$. Now, since $\mathrm{end}(s') = \mathrm{end}(s) < \mathrm{start}(t) \leq \mathrm{start}(t')$, we have $u' \in \mathrm{seq}(S',T')$ and $\mathrm{start}(u) \leq \mathrm{start}(u')$ and $\mathrm{end}(u') = \mathrm{end}(u)$.

This means that there exists some $u'' \in U$ with $\text{start}(u) \leq \text{start}(u'')$ and $\text{end}(u'') = \text{end}(u)$, which satisfies the second constraint in the definition of *rem*.

Finally, $rem(\text{seq}(S', T'), U)$ ensures that all instances in U have different end times. Together, this gives $rem(\text{seq}(S, T), U)$.

5. Assume $rem(\text{tim}(S', \tau), U)$. For any $u \in U$ we have $u \in \text{tim}(S', \tau)$ and thus $u \in S'$ and $\text{end}(u) - \text{start}(u) \leq \tau$. By the subset requirement in the definition of *rem*, we have $u \in S$ which means that $u \in \text{tim}(S, \tau)$ and thus $U \subseteq \text{tim}(S, \tau)$.

Next, take an arbitrary $u \in \text{tim}(S, \tau)$. Then $u \in S$ and there exists an $u' \in S'$ with $\text{start}(u) \leq \text{start}(u')$, $\text{end}(u') = \text{end}(u)$. Since $\text{end}(u) - \text{start}(u) \leq \tau$, we have $\text{end}(u') - \text{start}(u') \leq \tau$ and thus $u' \in \text{tim}(S', \tau)$. According to the def of *rem*, this means that there exists some $u'' \in U$ with $\text{start}(u') \leq \text{start}(u'')$, $\text{end}(u'') = \text{end}(u')$. Since this means that $\text{start}(u) \leq \text{start}(u'')$, $\text{end}(u'') = \text{end}(u)$ the second constraint in the definition of *rem* is satisfied.

Finally, $rem(\text{tim}(S', \tau), U)$ ensures that all instances in U have different end times. Together, this gives $rem(\text{tim}(S, \tau), U)$.

$\square$

5 An event detection algorithm

In this section, we present an imperative algorithm that, for a given event expression E, computes an event stream S for which $rem(\llbracket E \rrbracket, S)$ holds. Throughout this section, E denotes the event expression that is to be detected. The numbers $1 \ldots m$ are assigned to the subexpressions of E in bottom-up order, and we let E^i denote subexpression number i. Consequently, we have $E^m = E$ and $E^1 \in \mathcal{P}$.

Figure 2 presents the algorithm. The algorithm is executed once every time instant, and computes the current instance of E from the current instances of the primitive events, and from stored information about the past.

Each operator occurrence in the expression requires its own state variables, and thus variables are indexed from 1 to m. The variable a_i is used to store the current instance of E^i, and thus a_m contains the output of the algorithm after each execution. The auxiliary variables l_i, r_i, t_i and q_i store information about the past needed to detect E^i properly. In l_i and r_i, a single event instance is stored, t_i stores a time instant and q_i contains a set of event instances. The symbol $\langle \rangle$ is used to represent a non-occurrence, and we define $\text{start}(\langle \rangle) = \text{end}(\langle \rangle) = -1$ to simplify the algorithm.

The algorithm is designed for detection of arbitrary expressions, and the the main loop selects dynamically which part of the algorithm to execute for each subexpression. For systems where the event expressions of interest are static and known at the time of development, the main loop can be unrolled and the top-level conditionals, as well as all indices, can be statically determined. A concrete example of this is given in Figure 3.

5.1 Algorithm correctness

Next, the relation between this algorithm and the algebra semantics described in previous sections must be established. For this purpose, we need to formalise the algorithm output by constructing corresponding event streams.

Definition 5.1 *For $1 \leq i \leq m$, define*

$$\mathcal{A}(i) = \{e \mid e \in \text{out}(i, \tau) \wedge e \neq \langle \rangle \wedge \tau \in \mathcal{T}\}$$

```
for i from 1 to m
    if E^i ∈ P then
        a_i := the current instance of E^i, or ⟨⟩ if there is none.
    if E^i = E^j ∨ E^k then
        if start(a_j) ≤ start(a_k) then a_i := a_k else a_i := a_j
    if E^i = E^j + E^k then
        if start(l_i) < start(a_j) then l_i := a_j
        if start(r_i) < start(a_k) then r_i := a_k
        if l_i = ⟨⟩ or r_i = ⟨⟩ or (a_j = ⟨⟩ and a_k = ⟨⟩) then a_i := ⟨⟩
            else if start(a_k) ≤ start(a_j)
                then a_i := a_j ∪ r_i
                else a_i := l_i ∪ a_k
    if E^i = E^j − E^k then
        if t_i < start(a_k) then t_i := start(a_k)
        if t_i < start(a_j) then a_i := a_j else a_i := ⟨⟩
    if E^i = E^j ; E^k then
        a_i := ⟨⟩
        if a_k ≠ ⟨⟩ then
            foreach e in q_i
                if end(e) < start(a_k) and start(a_i) < start(e)
                then a_i := e
            if a_i ≠ ⟨⟩ then a_i := a_k ∪ a_i
        if t_i < start(a_j) then
            q_i := q_i ∪ {a_j}
            t_i := start(a_j)
    if E^i = (E^j)_τ then
        if end(a_j) − start(a_j) ≤ τ then a_i := a_j else a_i := ⟨⟩
```

Figure 2: For a given event expression E this algorithm computes an event stream S for which $rem(\llbracket E \rrbracket, S)$ holds. Initially, $t_i = -1$, $l_i = r_i = \langle\rangle$ and $q_i = \emptyset$ for $1 \leq i \leq m$.

$$
\begin{array}{|l|}
\hline
a_1 := \text{the current instance of } \mathsf{T}, \text{ or } \langle\rangle \text{ if there is none.} \\
a_2 := \text{the current instance of } \mathsf{P}, \text{ or } \langle\rangle \text{ if there is none.} \\
\text{if } \mathrm{start}(a_1) \leq \mathrm{start}(a_2) \text{ then } a_3 := a_2 \text{ else } a_3 := a_1 \\
a_4 := \text{the current instance of } \mathsf{B}, \text{ or } \langle\rangle \text{ if there is none.} \\
\text{if } t_5 < \mathrm{start}(a_4) \text{ then } t_5 := \mathrm{start}(a_4) \\
\text{if } t_5 < \mathrm{start}(a_3) \text{ then } a_5 := a_3 \text{ else } a_5 := \langle\rangle \\
\hline
\end{array}
$$

Figure 3: Statically simplified algorithm for detecting $(\mathsf{T}\vee\mathsf{P})-\mathsf{B}$. Initially, $t_5 = -1$.

where $\mathrm{out}(i,\tau)$ *denotes the value of* a_i *after executing the algorithm at times* 0 *to* τ.

We also introduce what can be thought of as a pointwise restriction relation, and a lemma that captures how it relates to the ordinary restriction policy.

Definition 5.2 *For an event instance* e, *an event stream* S *and* $\tau \in \mathcal{T}$, *define* $valid(e, S, \tau)$ *to hold if:*

$$
(e \in S \wedge \mathrm{end}(e) = \tau \wedge \neg\exists s(s \in S \wedge \mathrm{start}(e) < \mathrm{start}(s) \wedge \mathrm{end}(s) = \tau)) \vee
$$
$$
(e = \langle\rangle \wedge \neg\exists s(s \in S \wedge \mathrm{end}(s) = \tau))
$$

Lemma 5.1 *For event instances* $e_0, e_1, e_2, \ldots$ *and an event stream* S *such that* $valid(e_\tau, S, \tau)$ *holds for any* $\tau \in \mathcal{T}$, *let* $S' = \{e_0, e_1, e_2, \ldots\} - \{\langle\rangle\}$. *Then* $rem(S, S')$ *holds.*

Proof: By the definition of *valid*, it follows that $S' \subseteq S$. Next, take an arbitrary $s \in S$, and let $\tau = \mathrm{end}(s)$. Since $valid(e_\tau, S, \tau)$, we must have $e_\tau \neq \langle\rangle$, and thus $e_\tau \in S'$. From the definition of *valid*, we know that $\mathrm{start}(s) \leq \mathrm{start}(e_\tau)$. We also have $\mathrm{end}(e_\tau) = \mathrm{end}(s)$, which means that the second requirement in the definition of *rem* is satisfied. Finally, all elements in S' have different end times. Together, this implies that $rem(S, S')$ holds. $\qquad\square$

The correctness proof is organised as follows. For each of the six operators, we prove a lemma showing that for an expression of that type, the output is always valid and the internal state is correctly updated, with respect to the instances detected for the subexpresions. Finally, these lemmas are combined with Theorem 4.6 from the previous section to prove the algorithm correct.

Before turning to the operators, some general observations regarding the algorithm can be made. It is straightforward to see that during the ith iteration of the loop only variables with index i are changed, and all variables that are used have indices less then or equal to i, since the subexpressions of E are numbered in bottom-up order. Thus, when proving correctnes for subexpression E^i, it is sufficient to consider the ith iteration. The auxiliary predicates defined in this section do not refer to variables of index highter than i, and thus if they hold after the ith iteration, they will hold after iterations $i+1$ to m as well.

Disjunction: $E^i = E^j \vee E^k$

The disjunction operator is fairly simple and requires no auxiliary variables. If E^j and E^k occur at the same time, the restriction policy requires that the one with latest start time is selected. When the start times are the same, this implementation gives precedence to the right subexpression.

The fact that the implementation of the disjunction operator corresponds to the declarative semantics with restriction, with respect to the instances that are detected for the subexpressions, is formulated by the following lemma.

Lemma 5.2 *For $1 \leq i \leq m$ such that $E^i = E^j \vee E^k$, and any $\tau \in \mathcal{T}$, the following holds:*

i) $valid(a_i, \mathrm{dis}(\mathcal{A}(j), \mathcal{A}(k)), \tau)$ holds after executing the algorithm at time τ.

ii) $rem(\mathrm{dis}(\mathcal{A}(j), \mathcal{A}(k)), \mathcal{A}(i))$.

Proof:

i) If one or both of a_j and a_k are $\langle\rangle$, then trivially $valid(a_i, \mathrm{dis}(\mathcal{A}(j), \mathcal{A}(k)), \tau)$ holds after executing the disjunction part of the algorithm. Otherwise, both a_j and a_k belong to $\mathrm{dis}(\mathcal{A}(j), \mathcal{A}(k))$, and thus the one with maximum start time satisfies the condition of *valid*. If the start times are equal, the algorithm selects a_k, which satisfies the condition.

ii) Follows from *i)* and Lemma 5.1.

$\square$

Conjunction: $E^i = E^j + E^k$

For conjunctions, it is necessary to store the instance with maximum start time so far from each of the two subexpressions. This is formalised by the following predicate, which holds at the start of time instant τ if l_i and r_i have correct values.

Definition 5.3 *For $1 \leq i \leq m$ such that $E^i = E^j + E^k$, and for $\tau \in \mathcal{T}$, we define $constate(i, \tau)$ to hold if the following holds:*

- *l_i is an element in $\{e \mid e \in \mathcal{A}(j) \wedge \mathrm{end}(e) < \tau\}$ with maximum start time, or $\langle\rangle$ if that set is empty.*

- *r_i is an element in $\{e \mid e \in \mathcal{A}(k) \wedge \mathrm{end}(e) < \tau\}$ with maximum start time, or $\langle\rangle$ if that set is empty.*

The following lemma states that the conjunction operator is correctly implemented by the algorithm.

Lemma 5.3 *For $1 \leq i \leq m$ such that $E^i = E^j + E^k$, and any $\tau \in \mathcal{T}$, the following holds:*

i) $constate(i, \tau)$ holds at the start of time τ.

ii) $valid(a_i, \mathrm{con}(\mathcal{A}(j), \mathcal{A}(k)), \tau)$ holds after executing the algorithm at time τ.

iii) $rem(\mathrm{con}(\mathcal{A}(j), \mathcal{A}(k)), \mathcal{A}(i))$.

Proof:

i) $constate(i, 0)$ holds for an initial state where $l_i = r_i = \langle\rangle$. Next, assume that $constate(i, \tau)$ holds at the start of time τ. Then the first conditional in the conjunction part of the algorithm ensures that l_i contains an instance consistent with $constate(i, \tau+1)$, after executing the conjunction part. Similarly, the second conditional ensures the correctness of r_i. By induction, the lemma holds for any $\tau \in \mathcal{T}$.

ii) From the proof of *i)*, we know that $constate(i, \tau+1)$ holds after executing the first two conditionals of the conjunction part. If the guard of the third conditional is satisfied, it trivially follows that there is no instance in $\mathrm{con}(\mathcal{A}(j), \mathcal{A}(k))$ with end time τ, and thus the lemma holds after assigning $\langle\rangle$ to a_i. If the guard is false, we identify three separate cases. For the case

16

when $a_j = \langle\rangle$, we know that $l_i \cup a_k \in \mathrm{con}(\mathcal{A}(j), \mathcal{A}(k))$. Assume the existence of $e \in \mathrm{con}(\mathcal{A}(j), \mathcal{A}(k))$ with $\mathrm{start}(l_i \cup a_k) < \mathrm{start}(e)$ and $\mathrm{end}(e) = \tau$. Then, as $a_j = \langle\rangle$, we must have $e = e' \cup a_k$ with $e' \in \mathcal{A}(j)$, $\mathrm{start}(l_i) < \mathrm{start}(e')$ and $\mathrm{end}(e') < \tau$. This contradicts $constate(i, \tau+1)$, and thus no such e' exists which means that $l_i \cup a_k$ is valid. Since $a_j = \langle\rangle$, the inner conditional evaluates to false and $l_i \cup a_k$ is assigned to a_i, meaning that the lemma holds for this case. Similarly, for the case when $a_k = \langle\rangle$ the lemma holds after after assigning $a_j \cup r_i$ to a_i. The third case, when neither a_j nor a_k are $\langle\rangle$, both $a_j \cup r_i$ and $l_i \cup a_k$ belong to $\mathrm{con}(\mathcal{A}(j), \mathcal{A}(k))$. Using the same reasongin as in the previous cases, we have that there can exist no $e \in \mathrm{con}(\mathcal{A}(j), \mathcal{A}(k))$ with $\mathrm{start}(l_i \cup a_k) < \mathrm{start}(e)$, $\mathrm{start}(a_j \cup r_i) < \mathrm{start}(e)$ and $\mathrm{end}(e) = \tau$. If the inner conditional holds, we have $\mathrm{start}(a_k) \leq \mathrm{start}(a_j)$ and by $constate(i, \tau+1)$ we also have $\mathrm{start}(a_k) \leq \mathrm{start}(r_i)$. Thus $\mathrm{start}(l_i \cup a_k) \leq \mathrm{start}(a_j \cup r_i)$, and the lemma holds after after assigning $a_j \cup r_i$ to a_i. Similarly, if the inner conditional does not hold, the lemma holds after after assigning $l_i \cup a_k$ to a_i.

  iii) Follows from *ii)* and Lemma 5.1.

<div style="text-align:right">□</div>

Negation: $E^i = E^j - E^k$

According to the semantics of the negation operator, an instance of B is an instance of $B - C$ unless it is invalidated by some instance of C occurring within its interval. If the current instance of B is invalidated at all, it is invalidated by the instance of C with maximum start time (of those that have occured so far). Thus, it is sufficient to store a single start time, since the end time is trivially known to be less than the end time of the current instance of B.

Definition 5.4 *For $1 \leq i \leq m$ such that $E^i = E^j - E^k$, and for $\tau \in \mathcal{T}$, we define negstate(i, τ) to hold if t_i is the maximum start time of the elements in $\{e \mid e \in \mathcal{A}(k) \wedge \mathrm{end}(e) < \tau\}$, or -1 if this set is empty.*

Lemma 5.4 *For $1 \leq i \leq m$ such that $E^i = E^j - E^k$, and any $\tau \in \mathcal{T}$, the following holds:*

  i) negstate(i, τ) holds at the start of time τ.

  ii) valid$(a_i, \mathrm{neg}(\mathcal{A}(j), \mathcal{A}(k)), \tau)$ holds after executing the algorithm at time τ.

  iii) rem$(\mathrm{neg}(\mathcal{A}(j), \mathcal{A}(k)), \mathcal{A}(i))$.

Proof:

  i) *negstate$(i, 0)$* holds for an initial state where $t_i = -1$. Next, assume that *negstate(i, τ)* holds at the start of time τ. Then the first conditional in the negation part of the algorithm ensures that t_i contains the value specified by *negstate$(i, \tau+1)$* after executing the negation part. By induction, the lemma holds for any $\tau \in \mathcal{T}$.

  ii) From the proof of *i)*, we know that *negstate$(i, \tau+1)$* holds after executing the first conditional of the negation part. If the guard of the second conditional holds, then we have $a_j \neq \langle\rangle$ and thus $a_j \in \mathcal{A}(j)$. According to *negstate$(i, \tau+1)$* there is no e in $\mathcal{A}(k)$ with $\mathrm{start}(a_j) \leq \mathrm{start}(e)$ and $\mathrm{end}(e) < \mathrm{end}(a_j) = \tau$, and thus $a_j \in \mathrm{neg}(\mathcal{A}(j), \mathcal{A}(k))$. Trivially, since a_j is the only instance in $\mathcal{A}(j)$ with end time τ, we have *valid$(a_j, \mathrm{neg}(\mathcal{A}(j), \mathcal{A}(k)))$*. Thus, the lemma holds after assigning a_j to a_i.

<div style="text-align:center">17</div>

iii) Follows from *ii)* and Lemma 5.1.

$\square$

Sequence: $E^i = E^j ; E^k$

The sequence operator requires the most complex algorithm. The reason for this is that in order to detect a sequence $B;C$ correctly, we must store several instances of B. Once C occurs, the start time of that instance determines with which of the stored instances of B it should be combined to form the instance of $B;C$.

Definition 5.5 *For $1 \leq i \leq m$ such that $E^i = E^j ; E^k$, and for $\tau \in \mathcal{T}$, we define seqstate(i, τ) to hold if the following holds:*

- t_i *is the maximum start time of the elements in $\{e \mid \mathcal{A}(j) \wedge \text{end}(e) < \tau\}$, or -1 if this set is empty.*

- $q_i = \{e \mid \mathcal{A}(j) \wedge \text{end}(e) < \tau \wedge \neg \exists e'(e' \in \mathcal{A}(j) \wedge e' \neq e \wedge \text{start}(e) \leq \text{start}(e') \wedge \text{end}(e') \leq \text{end}(e))\}$

Lemma 5.5 *For $1 \leq i \leq m$ such that $E^i = E^j ; E^k$, and any $\tau \in \mathcal{T}$, the following holds:*

i) *seqstate(i, τ) holds at the start of time τ.*

ii) *valid$(a_i, \text{seq}(\mathcal{A}(j), \mathcal{A}(k)), \tau)$ holds after executing the algorithm at time τ.*

iii) *rem$(\text{seq}(\mathcal{A}(j), \mathcal{A}(k)), \mathcal{A}(i))$.*

Proof:

i) *seqstate$(i, 0)$* holds for an initial state where $t_i = -1$ and $q_i = \emptyset$. Next, assume that *seqstate(i, τ)* holds at the start of time τ. The first conditional of the sequence part of the algorithm does not change the values of t_i and q_i. If the second conditional holds, t_i is updated to the value specified by *seqstate$(i, \tau + 1)$*. Also, by *seqstate(i, τ)*, we know that there is no $e \in \mathcal{A}(j)$ with $t_i < \text{start}(e)$ and $\text{end}(e) < \tau$, which implies that *seqstate$(i, \tau + 1)$* holds after adding a_j to q_i. If the second conditional does not hold, no changes are required for *seqstate$(i, \tau + 1)$* to hold. By induction, the lemma holds.

ii) From *i)*, we know that *seqstate(i, τ)* holds at the start of time τ. Consider first the case when $a_k = \langle \rangle$. Then there is no instance in $\text{seq}(\mathcal{A}(j), \mathcal{A}(k))$ with end time τ. Thus, the lemma holds after assigning $\langle \rangle$ to a_i. In the second case we have $a_k \neq \langle \rangle$. If $a_i = \langle \rangle$ after executing the foreach statement, then there is no instance e in q_i with $\text{end}(e) < \text{start}(a_k)$, and thus by *seqstate$(i, \tau)$* there is no $e \in \mathcal{A}(j)$ with $\text{end}(e) < \text{start}(a_k)$. This implies that there is no $e' \in \text{seq}(\mathcal{A}(j), \mathcal{A}(k))$ with $\text{end}(e') = \tau$, and thus the lemma holds after assigning $\langle \rangle$ to a_i. If $a_i \neq \langle \rangle$ after executing the foreach statement, we have $\text{end}(a_i) < \text{start}(a_k)$ and thus $a_i \cup a_k \in \text{seq}(\mathcal{A}(j), \mathcal{A}(k))$. By *seqstate$(i, \tau)$* we also know that there is no $e' \in \mathcal{A}(j)$ with $\text{start}(a_i) < \text{start}(e')$ and $\text{end}(e') < \text{start}(a_k)$, and thus *valid$(a_i \cup a_k, \text{seq}(\mathcal{A}(j), \mathcal{A}(k))$* and the lemma holds after assigning $a_i \cup a_k$ to a_i.

iii) Follows from *ii)* and Lemma 5.1.

$\square$

Temporal restriction: $E^i = (E^j)_{\tau'}$

The temporal restriction is fairly straightforward to implement and requires no auxiliary state variables.

Lemma 5.6 *For $1 \leq i \leq m$ such that $E^i = (E^j)_{\tau'}$, and any $\tau \in \mathcal{T}$, the following holds:*

   i) $valid(a_i, \mathrm{tim}(\mathcal{A}(j), \mathcal{A}(k)), \tau)$ holds after executing the algorithm at time τ.

   ii) $rem(\mathrm{tim}(\mathcal{A}(j), \mathcal{A}(k)), \mathcal{A}(i))$.

Proof:

   i) If $a_j = \langle \rangle$, lemma holds after assigning $\langle \rangle$ to a_i, which is done in both branches of the conditional. If $a_j \neq \langle \rangle$ and the conditional holds, we have $a_j \in \mathrm{tim}(\mathcal{A}(j), \tau')$. Since a_j is the only instance in $\mathcal{A}(j)$ with end time τ, it follows that the lemma holds after assigning a_j to a_i. If the conditional is false, there is no instance in $\mathrm{tim}(\mathcal{A}(j), \tau')$ with end time τ, so the lemma holds after assigning $\langle \rangle$ to a_i.

   ii) Follows from *i)* and Lemma 5.1.

$\square$

Putting it all together

The following theorem establishes the correctness of the algorithm by stating that for each subexpression E^i, including E itself, the detected instances correspond to a valid restriction of $[\![E^i]\!]$.

Theorem 5.1 *For any i such that $1 \leq i \leq m$, $rem([\![E^i]\!], \mathcal{A}(i))$ holds.*

Proof: For $E^i \in \mathcal{P}$, we have $\mathcal{A}(i) = [\![E^i]\!]$ under the assumption that the interpretation correctly represents the real-world scenario. Thus $rem([\![E^i]\!], \mathcal{A}(i))$ holds trivially.

   Next, assume that for some i, $rem([\![E^x]\!], \mathcal{A}(x))$ holds for any $1 \leq x < i$. If $E^i = E^j \vee E^k$, then according to Lemma 5.2 we have $rem(\mathrm{dis}(\mathcal{A}(j), \mathcal{A}(k)), \mathcal{A}(i))$. Since the subexpressions are numbered bottom-up, we have $j < i$ and $k < i$, so by assumption $rem([\![E^j]\!], \mathcal{A}(j))$ and $rem([\![E^k]\!], \mathcal{A}(k))$ holds. Then, according to Theorem 4.6, $rem(\mathrm{dis}([\![E^j]\!], [\![E^k]\!]), \mathcal{A}(i))$ holds, which means that $rem([\![E^i]\!], \mathcal{A}(i))$ holds. A similar proof can be constructed for each of the operators. By induction, the theorem holds. $\square$

5.2 Memory complexity

Instances are not of a fixed size, but an instance from a subexpression of E contains at most $m/2$ primitive instances, one from each identifier occurrence in E. Thus, assuming that the elements in the value domains are of constant size, the size of a single event instance is bounded.

   A quick analysis of the algorithm reveals that each disjunction, conjunction, negation and temporal restriction in the event expression requires a limited amount of storage. The storage required for a sequence operator depend on the maximum size of q_i, for which no bound exists in the general case. For an important class of sequence expressions, however, the detection algorithm can be redefined to ensure limited memory and time complexity.

```
    foreach e in q_i
        if end(e) < τ^c − τ' then q_i := q_i − {e};  l_i := e
    a_i := ⟨⟩
    if a_k ≠ ⟨⟩ then
        foreach e in q_i
            if end(e) < start(a_k) and start(a_i) < start(e) then a_i := e
        if a_i = ⟨⟩ then a_i := l_i
        if a_i ≠ ⟨⟩ then a_i := a_k ∪ a_i
    if t_i < start(a_j) then q_i := q_i ∪ {a_j};  t_i := start(a_j)
```

Figure 4: Algorithm for $E^i = E^j ; E^k$ when $E^k \equiv E^k_{\tau'}$

For a sequence $A;B$ where we know that the maximum length of the instances of B is τ, which can be expressed as $B \equiv B_\tau$, this limits the number of instances of A that must be stored in order to detect the sequence correctly. Informally, the start of any instance of B will be at most τ time units back in time, and thus there is no need to store more than one instance of A that ends earlier than this, if we store one with maximum start time. From the instances of A that end later than this point in time, we need to store several, as in the original algorithm.

The improved algorithm for detecting $A;B$ when $B \equiv B_\tau$ with bounded memory is presented in Figure 4. Here, τ^c is used to access the current time instant. The state is similar to the state used for sequences in the original algorithm, but this q_i contains a suffix of the q_i variable of the original version. From the remaining elements, a single element with maximum start time is stored in l_i. Since the size of q_i never exceeds $\tau+1$, this type of sequences can be detected with limited memory.

For this new algorithm, we prove a lemma similar to those in the previous section.

Definition 5.6 *For $1 \leq i \leq m$ such that $E^i = E^j ; E^k$ and $E^k \equiv E^k_{\tau'}$, and for $\tau \in \mathcal{T}$, we define $newstate(i, \tau)$ to hold if the following holds:*

- *t_i is the maximum start time of the elements in $\{e \mid \mathcal{A}(j) \wedge \text{end}(e) < \tau\}$, or -1 if this set is empty.*

- *$q_i = \{e \mid \mathcal{A}(j) \wedge \text{end}(e) < \tau \wedge \tau - \tau' - 1 \leq \text{end}(e) \wedge \neg \exists e'(e' \in \mathcal{A}(j) \wedge e' \neq e \wedge \text{start}(e) \leq \text{start}(e') \wedge \text{end}(e') \leq \text{end}(e))\}$*

- *l_i is an element in $\{e \mid \mathcal{A}(j) \wedge \text{end}(e) < \tau - \tau' - 1\}$ with maximum start time, or $\langle\rangle$ if that set is empty.*

Lemma 5.7 *For $1 \leq i \leq m$ such that $E^i = E^j ; E^k$ and $E^k \equiv E^k_{\tau'}$, and any $\tau \in \mathcal{T}$, the following holds:*

- *i) $newstate(i, \tau)$ holds at the start of time τ.*

- *ii) $valid(a_i, \text{seq}(\mathcal{A}(j), \mathcal{A}(k)), \tau)$ holds after executing the algorithm at time τ.*

- *iii) $rem(\text{seq}(\mathcal{A}(j), \mathcal{A}(k)), \mathcal{A}(i))$.*

Proof:

i) $newstate(i, 0)$ holds for an initial state where $t_i = -1$, $q_i = \emptyset$ and $l_i = \langle\rangle$. Next, assume that $newstate(i, \tau)$ holds at the start of time τ. If the conditional of the first foreach statement holds for e, we have $\text{end}(e) = \tau - \tau' - 1$. This means that the conditional can hold for at most one element of q_i. The definition of

newstate requires this e to be removed from q_i in order for $newstate(i, \tau+1)$ to hold. By $newstate(i, \tau)$, we also have that e fulfills the requirement on l_i as specified by $newstate(i, \tau+1)$. In the rest of the algorithm, q_i is updated in the same way as for the original sequence algorithm, and in the end $newstate(i, \tau+1)$ holds. Then by induction the lemma holds for any $\tau \in \mathcal{T}$.

ii) From *i)*, we know that $newstate(i, \tau)$ holds at the start of time τ.

The case when $a_k = \langle \rangle$ follows the proof for the original sequence. In the case when $a_k \neq \langle \rangle$, if $a_i \neq \langle \rangle$ after executing the second foreach statement, then we have $valid(a_i \cup a_k, \text{seq}(\mathcal{A}(j), \mathcal{A}(k))$ as in the proof of the original sequence. If $a_i = \langle \rangle$ after executing the second foreach statement, then there is no instance e in q_i with $\text{end}(e) < \text{start}(a_k)$. In this case, we assign l_i to a_i. If $l_i \neq \langle \rangle$ we know that $\text{end}(l_i) < \text{start}(a_k)$, and then since l_i has the value specified by $newstate(i, \tau+1)$, we have that $valid(l_i \cup a_k, \text{seq}(\mathcal{A}(j), \mathcal{A}(k)))$ holds.

Thus, arriving at the next conditional we know that either $a_i = \langle \rangle$ and there is no instance e in $\mathcal{A}(j)$ with $\text{end}(e) < \text{start}(a_k)$, or that $valid(a_i \cup a_k, \text{seq}(\mathcal{A}(j), \mathcal{A}(k)))$ holds. Thus, the lemma holds after this conditional, and this is not affected by the final conditional.

iii) Follows from *ii)* and Lemma 5.1.

$\square$

5.3 Time complexity

As a result of instances not having a fixed size, assigning an instance to a variable might not be a constant operation, but rather proportional to the instance size. Thus, each operator contributes with at least a factor m to the complexity for the whole algorithm. For sequences, a straightforward representation of the q_i variables gives a linear time complexity for finding the best matching instance, with respect to the size limit of that q_i variable. This gives a a total complexity of $O(mn')$, where m is the number of subexpressions in E, $n' = \max(m, n)$ and n is the maximum size limit of the q_i variables.

Due to the particular characteristics of q_i, a more elaborate implementation is possible, where we keep q_i sorted with respect to end times. Since q_i should contain no fully overlapping instances, this means thay it will be sorted with respect to start times as well. Since an instance that is added to q_i has a later start time than the instances already in q_i, and elements are removed when they become too old, q_i will behave like a first-in-first-out queue.

Consequently, keeping q_i updated is only bounded by the m factor for variable-sized instances. However, when an instance of B occurs we need to find the best matching instance in q_i efficiently, i.e., the instance with latest start time among those that end before the start time of the B instance. Since q_i is sorted with respect to both start and end times, this can be implemented as a straightforward binary search if the implementation of q_i allows constant access to arbitrary elements.

To allow this, we base the implementation of q_i on a static array, and use two variables to mark the part of this array that currently contain q_i. When elements are added and removed, these variables are increased accordingly, and when the end of the array is reached they simply continue at the beginning. Since the size limit of q_i is known, using the same size for the array ensures that there is always room for q_i in the array, i.e., that the start marker will not overtake the end marker. Using this implementation, the total complexity is $O(mn'')$, where m is the number of subexpressions in E, $n'' = \max(m, \log n)$ and n is the maximum size limit of the q_i variables.

6   Transformation algorithm

This section describes how event expressions can be automatically transformed into equivalent expressions that allow a more efficient detection. The transformation algorithm is based on the algebraic laws describing how temporal restrictions can be propagated through an expression, presented in Theorem 4.4.

To simplify the presentation, we extend the algebra syntax with two constructs. The symbol ∞ is added to the temporal domain to allow temporally restricted and unrestricted expressions to be treated uniformly. Formally, we define $A_\infty = A$. Since the improved sequence algorithm is defined for sequences $A;B$ where $B \equiv B_\tau$, we introduce the notation $A;_\tau B$ to label sequences with this information.

The transformation algorithm is based on a recursive function that takes an expression and a time as input, and returns the transformed expression and a time. This function is defined in Figure 5. The input time represents a temporal restric-

$$\mathrm{transform}(A, \tau) = \langle A, 0 \rangle \qquad \text{if } A \in \mathcal{P}$$

$$\mathrm{transform}(A \vee B, \tau) = \langle A' \vee B', \max(\tau_a, \tau_b) \rangle$$
$$\text{where } \langle A', \tau_a \rangle = \mathrm{transform}(A, \tau)$$
$$\langle B', \tau_b \rangle = \mathrm{transform}(B, \tau)$$

$$\mathrm{transform}(A + B, \tau) = \langle A' + B', \infty \rangle$$
$$\text{where } \langle A', \tau_a \rangle = \mathrm{transform}(A, \tau)$$
$$\langle B', \tau_b \rangle = \mathrm{transform}(B, \tau)$$

$$\mathrm{transform}(A - B, \tau) = \langle A' - B', \tau_a \rangle$$
$$\text{where } \langle A', \tau_a \rangle = \mathrm{transform}(A, \tau)$$
$$\langle B', \tau_b \rangle = \mathrm{transform}(B, \min(\tau_a, \tau))$$

$$\mathrm{transform}(A;B, \tau) = \begin{cases} \langle A';_{\tau_b} B', \infty \rangle & \text{if } \tau_b \leq \tau \\ \langle A';_\tau (B'_\tau), \infty \rangle & \text{if } \tau < \tau_b \end{cases}$$
$$\text{where } \langle A', \tau_a \rangle = \mathrm{transform}(A, \tau)$$
$$\langle B', \tau_b \rangle = \mathrm{transform}(B, \tau)$$

$$\mathrm{transform}(A_{\tau'}, \tau) = \begin{cases} \langle A', \tau_a \rangle & \text{if } \tau_a \leq \tau'' \\ \langle A'_{\tau''}, \tau'' \rangle & \text{if } \tau'' < \tau_a \end{cases}$$
$$\text{where } \langle A', \tau_a \rangle = \mathrm{transform}(A, \tau'')$$
$$t'' = \min(\tau, \tau')$$

Figure 5: Definition of the transformation function

tion that can be applied to the expression without changing the meaning of the expression as a whole. The returned time represents a temporal restriction that can be applied to the transformed expression without changing its meaning.

This informal description is formalised in the following lemma, which states that the transformation function preserves the semantics of the original expression when called properly. It also defines the meaning of the returned time.

Lemma 6.1 *For an event expression E and $\tau \in \mathcal{T}$, if $\mathrm{transform}(E, \tau) = \langle E', \tau' \rangle$, then $E_\tau \equiv E'_\tau$ and $E' \equiv E'_{\tau'}$.*

Proof: For each case in the definition of transform, we assume that the recursive calls are correct, and show that this implies that the return tuple is correct. Since the recursion trivially terminates, the lemma holds by induction. In the following

proof, $\equiv^{23}$ denotes that the equivalence follows from law number 23 in Theorem 4.4, etc. and $\equiv^{\mathrm{a}}$ indicates that the equivalence follows from the assumptions.

Primitive For $E \in \mathcal{P}$, we have $E \equiv^{23} E_0$. Thus, $\langle A, 0 \rangle$ is a valid answer.

Disjunction For $E = A \vee B$, we have $A_\tau \equiv A'_\tau$, $A' \equiv A'_{\tau_a}$, $B_\tau \equiv B'_\tau$ and $B' \equiv B'_{\tau_b}$ since we assume that the lemma holds for the two recursive calls. Then, we have $(A \vee B)_\tau \equiv^{25} A_\tau \vee B_\tau \equiv^{\mathrm{a}} A'_\tau \vee B'_\tau \equiv^{25} (A' \vee B')_\tau$. We also have $(A' \vee B') \equiv^{\mathrm{a}} A'_{\tau_a} \vee B'_{\tau_b} \equiv^{34} (A'_{\tau_a} \vee B'_{\tau_b})_{\max(\tau_a, \tau_b)} \equiv^{\mathrm{a}} (A' \vee B')_{\max(\tau_a, \tau_b)}$.

Thus, $\langle A' \vee B', \max(\tau_a, \tau_b) \rangle$ is a valid answer to $\mathrm{transform}(A \vee B, \tau)$.

Conjunction For $E = A + B$, we assume $A_\tau \equiv A'_\tau$, $A' \equiv A'_{\tau_a}$, $B_\tau \equiv B'_\tau$ and $B' \equiv B'_{\tau_b}$. Then, we have $(A + B)_\tau \equiv^{26,35} (A_\tau + B_\tau)_\tau \equiv^{\mathrm{a}} (A'_\tau + B'_\tau)_\tau \equiv^{26,35} (A' + B')_\tau$ and by definition $(A' + B')_\infty \equiv (A' + B')$ so $\langle A' + B', \infty \rangle$ is a valid answer.

Negation For $E = A - B$, we assume $A_\tau \equiv A'_\tau$, $A' \equiv A'_{\tau_a}$, $B_{\tau'} \equiv B'_{\min(\tau_a, \tau)}$ and $B' \equiv B'_{\tau_b}$. Then, we have $(A - B)_\tau \equiv^{27} A_\tau - B \equiv^{\mathrm{a}} A'_\tau - B \equiv^{\mathrm{a}} (A'_{\tau_a})_\tau - B \equiv^{24} A'_{\min(\tau_a, \tau)} - B \equiv^{27} (A' - B)_{\min(\tau_a, \tau)} \equiv^{28} (A' - B_{\min(\tau_a, \tau)})_{\min(\tau_a, \tau)} \equiv^{\mathrm{a}} (A' - B'_{\min(\tau_a, \tau)})_{\min(\tau_a, \tau)} \equiv^{28} (A' - B')_{\min(\tau_a, \tau)}$ and $(A' - B')_{\tau_a} \equiv^{27} (A'_{\tau_a} - B') \equiv^{\mathrm{a}} (A' - B')$ so $\langle A' - B', \tau_a \rangle$ is a valid answer.

Sequence For $E = A;B$, we assume $A_\tau \equiv A'_\tau$, $A' \equiv A'_{\tau_a}$, $B_\tau \equiv B'_\tau$ and $B' \equiv B'_{\tau_b}$. We consider the two cases separately.

If $\tau_b \leq \tau$, then $(A - B)_\tau \equiv^{29,30} (A_\tau - B_\tau)_\tau \equiv^{\mathrm{a}} (A'_\tau - B'_\tau)_\tau \equiv^{29,30} (A' - B')_\tau$ and trivially $(A' - B')_\infty \equiv A' - B'$. So $\langle A';B', \infty \rangle$ is a valid answer, and so is $\langle A';_{\tau_b} B', \infty \rangle$.

If $\tau < \tau_b$, then $(A - B)_\tau \equiv^{29,30} (A_\tau - B_\tau)_\tau \equiv^{\mathrm{a}} (A'_\tau - B'_\tau)_\tau \equiv^{29} (A' - B'_\tau)_\tau$ and trivially $(A' - (B'_\tau))_\infty \equiv (A' - (B'_\tau))$. So $\langle A';(B'_\tau), \infty \rangle$ is a valid answer, and so is $\langle A';_\tau (B'_\tau), \infty \rangle$.

Temporal For $E = A_{\tau'}$, we assume $A_{\tau''} \equiv A'_{\tau''}$ and $A' \equiv A'_{\tau_a}$. Let $t'' = \min(\tau, \tau')$, and consider the two cases:

If $\tau_a \leq \tau''$, then $\tau_a \leq \tau'$ and we have $(A_{\tau'})_\tau \equiv^{31} (A_\tau)_{\tau'} \equiv^{\mathrm{a}} (A'_\tau)_{\tau'} \equiv^{31} (A'_{\tau'})_\tau \equiv^{\mathrm{a}} ((A'_{\tau_a})_{\tau'})_\tau \equiv^{24} (A'_{\min(\tau_a, \tau')})_\tau \equiv (A'_{\tau_a})_\tau \equiv^{\mathrm{a}} A'_\tau$ and by assumption $A'_{\tau_a} \equiv A'$ so $\langle A', \tau_a \rangle$ is a valid answer.

If $\tau'' < \tau_a$, then $(A_{\tau'})_\tau \equiv^{24} ((A_{\tau'})_\tau)_\tau \equiv^{24} (A_{\min(\tau', \tau)})_\tau \equiv (A_{\tau''})_\tau \equiv^{\mathrm{a}} (A'_{\tau''})_\tau$ and $(A'_{\tau''})_{\tau''} \equiv^{24} A'_{\tau''}$ so $\langle A'_{\tau''}, \tau'' \rangle$ is a valid answer.

$\square$

Finally, this lemma is used to prove that the transformation preserves the semantic meaning of the expression, and that sequences are labeled correctly.

Theorem 6.1 *If E is an event expression and $\mathrm{transform}(E, \infty) = \langle E', \tau' \rangle$, then $E \equiv E'$ holds. Also, all sequences in E' are on the labeled form $A;_\tau B$, where $B \equiv B_\tau$ holds.*

Proof: From Lemma 6.1, $E \equiv E'$ follows trivially. A subexpression in E' on the form $A;_\tau B$ was created by the sequence part of the transformation algorithm, which has two cases. In the first case, $\langle B, \tau \rangle$ was the result of a call to transform, which according to Lemma 6.1 implies that $B \equiv B_\tau$ In the second case, $B = (B')_\tau$, which trivially means that $B \equiv B_\tau$. $\square$

The time complexity of the transformation algorithm is linear with respect to the size of E. If no sequence in E' is labeled with ∞, then E' (and consequently E) can be correctly detected with limited memory.

Example 6.1 $\text{transform}((\mathsf{B};\mathsf{B})_2-(\mathsf{P};(\mathsf{P}+\mathsf{T})), \infty) = \langle(\mathsf{B};_0\mathsf{B})_2-(\mathsf{P};_2(\mathsf{P}+\mathsf{T})_2), 2\rangle$ which means that this expression can be detected with limited memory. Note that the temporal restriction in the left subexpression of the negation has been propagated to the right subexpression, making it detectable with limited memory.

7 Conclusions and future work

We have presented a fully formal event algebra with operators for disjunction, conjunction, negation, sequence and temporal restriction. To allow an efficient implementation, a formal restriction policy was defined. This restriction policy is applied to the expression as a whole, rather than to the individual operator occurences, which means that a user of the algebra is not required to understand the effects of nested restrictions.

A number of algebraic laws were presented that facilitates formal reasoning and justifies the algebra semantics by showing to what extent the operators comply with intuition. When restriction is applied, these laws are valid with respect to the start and end time of the detected event instances. We presented an imperative algorithm that computes a restricted version of the event stream specified by the algebra semantics, in accordance with the restriction policy. For the user of the algebra, this means that at any time when there is one or more occurrences of the composite event, one of them will be detected by the algorithm. Finally, criteria under which detection can be performed with limited resources were identified, and we described an algorithm by which many expressions can be transformed to meet these criteria.

Our ongoing work includes investigating how to combine the algebra with languages that specifically target reactive systems, in particular Esterel [2], AFRP [11] and Timber [4]. We are also investigating how information about primitive event frequencies can be used to lower the size limits of the q_i variables, resulting in more precise worst case memory and time estimates.

References

[1] J. F. Allen and G. Ferguson. Actions and events in interval temporal logic. *Journal of Logic and Computation*, 4(5):531–579, Oct. 1994.

[2] G. Berry. *The Esterel-V5 Language Primer*. CMA and Inria, Sophia-Antipolis, France, v 5.21, release 2.0 edition, May 1999.

[3] J. Carlson and B. Lisper. An interval-based algebra for restricted event detection. In *First Int. Workshop on Formal Modeling and Analysis of Timed Systems (FORMATS 2003)*, Marseille, France, Sept. 2003.

[4] M. Carlsson, J. Nordlander, and D. Kieburtz. The semantic layers of Timber. In *Proceedings of the First Asian Symposium on Programming Languages and Systems (APLAS'2003)*, volume 2895 of *Lecture Notes in Computer Science*, Beijing, China, 26–29 Nov. 2003. Springer-Verlag.

[5] S. Chakravarthy and D. Mishra. Snoop: An expressive event specification language for active databases. *Data Knowledge Engineering*, 14(1):1–26, 1994.

[6] A. Galton and J. C. Augusto. Two approaches to event definition. In *Proc. of Database and Expert Systems Applications 13th Int. Conference (DEXA'02)*, volume 2453 of *Lecture Notes in Computer Science*. Springer-Verlag, Sept. 2002.

[7] S. Gatziu and K. R. Dittrich. Events in an active object-oriented database system. In *Proc. 1st Intl. Workshop on Rules in Database Systems (RIDS)*, Edinburgh, UK, Sept. 1993. Springer-Verlag.

[8] 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.

[9] R. A. Kowalski and M. J. Sergot. A logic-based calculus of events. *New Generation Computing*, 4:67–95, 1986.

[10] 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.

[11] H. Nilsson, A. Courtney, and J. Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop (HASKELL-02)*, pages 51–64, New York, Oct. 3 2002. ACM Press.

[12] C. Sánchez, S. Sankaranarayanan, H. Sipma, T. Zhang, D. Dill, and Z. Manna. Event correlation: Language and semantics. In *Embedded Software, Third International Conference, EMSOFT 2003*, volume 2855 of *Lecture Notes in Computer Science*, pages 323–33. Springer, 2003.