A resource-efficient event algebra

Jan Carlson and Björn Lisper

School of Innovation, Design and Engineering
Mälardalen University, Västerås, Sweden
{jan.carlson, bjorn.lisper}@mdh.se

Abstract

Events play many roles in computer systems, ranging from hardware interrupts, over event-based software architecture, to monitoring and managing of complex systems. In many applications, however, individual event occurrences are not the main point of concern, but rather the occurrences of certain event patterns. Such event patterns can be defined by means of an event algebra, i.e., expressions representing the patterns of interest are built from simple events and operators such as disjunction, sequence, etc.

We propose a novel event algebra with intuitive operators (a claim which is supported by a number of algebraic properties). We also present an efficient detection algorithm that correctly detects any expression with bounded memory, which makes this algebra particularly suitable for resource-constrained applications such as embedded systems.

Key words: Event detection, event notification, embedded systems

The final version of this article is: `doi:10.1016/j.scico.2010.06.010`

1 Introduction

The notion of events can appear in many different contexts in a computer system, often representing significant occurrences in the system environment, but also as a means of internal communication. For example, embedded systems are typically designed to react either to events generated by external sensors, or by periodically occurring timer events [1]. This event-driven execution model is also found in other types of applications, for example graphical user interfaces and web server programs.

On a higher level, large complex systems can be designed according to an event-based architectural style, where the communication between different parts of the system is based on a publish/subscribe interaction paradigm [2]. Event consumers express an interest in certain events by registering a subscription with an intermediary event manager. When an event is published, it is matched against the current subscriptions and relayed to the appropriate consumers.

For distributed systems, and in particular those consisting of heterogeneous subsystems (written in different programming languages, running on different hardware, etc.), the communication functionality can be structured as a separate middleware layer between the operating system and the applications [3], to hide low-level details related to distribution and the underlying operating system and hardware. Event-based middleware, e.g., Hermes [4] and READY [5], provide a uniform high-level interface of event related services, which allows seamless event handling also between heterogeneous subsystems.

On an even higher level, event handling is useful when managing, monitoring or exploring complex systems, including large software systems or networks but also real-world systems like stock markets or news report services. Here, a main concern is dealing effectively with very large volumes of event occurrences, and to filter out only those that are of interest in a particular situation. Examples of work in this category include monitoring of real-time systems [6], supervision of telecommunication networks [7] and air traffic control [8].

1.1 Event patterns and event algebras

In many applications, individual event occurrences are not the main point of concern, but rather the occurrences of certain event patterns. To support this, an event framework can provide means to specify event patterns, and allow these patterns to be used by the application in the same way as ordinary events. This means that the details of pattern detection are moved from the application to the event framework. For example, a subsystem might subscribe to the pattern *"A and B occur within 2 seconds"*, instead of subscribing to A and B, and perform the detection of the desired situation internally.

Event patterns can be defined in many ways, e.g., in some temporal or modal logic, by finite state machines, or as ordinary program code. Some techniques have high expressiveness, allowing a wide range of patterns to be defined. Other methods can only express a limited set of patterns, but instead provide very efficient detection of the patterns within this set.

Naturally, the nature of a certain domain determines how the tradeoff between expressiveness and efficiency should be chosen. We have focused on resource-constrained applications, such as embedded systems, where low and predictable resource usage is vital. Furthermore, we consider applications where usability and simplicity is favoured over high expressiveness.

When patterns grow in complexity, a compositional method is favourable, where complex patterns can be constructed by composing smaller patterns. One type of technique that achieves this is *event algebras*, where an event pattern is defined by an expression built recursively from atomic events and algebra operators. This approach is commonly used in languages for active databases, such as Snoop [9,10], Ode [11] and SAMOS [12,13], but also in some general, high-level event notification systems [14].

Common to many event specification methods is that they consider event pattern occurrences to be instantaneous, i.e., each occurrence is associated with a single time instant, normally the time at which it can be detected. As shown by Galton and Augusto [15], this results in unintended semantics for some operator combinations. As an example, consider the sequence operator, with the intuitive interpretation of $A;B$ being *"A occurs and then B occurs"*. With single point semantics, 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 $A;C$. Consequently, $B;(A;C)$ has exactly the same meaning as $A;(B;C)$, which does not match the intuitive meaning of sequential composition.

As a solution to this problem, Galton and Augusto propose that occurrences are associated with intervals rather than single time points, following the practice of knowledge representation techniques such as Event Calculus [16] and Interval Calculus [17]. Although this allows for a more intuitive operator semantics, it is not clear how this property can be preserved if we also impose significant resource constraints.

We propose an event algebra with a semantics based on time intervals, and show that it complies with algebraic laws that intuitively ought to hold for the algebra operators. To achieve resource efficiency, we define the semantics in two steps: a simple but inefficient operator semantics, and a formal restriction policy that specifies a subset of the simple semantics that should be detected. This allows any event expression to be correctly detected with bounded memory, while at the same time retaining the desired properties of the algebra operators. For an extended version of this paper, addressing also for example its impact on real-time schedulability analysis, see [18,19].

The paper is organised as follows: The event algebra is defined in Section 2, and Section 3 presents a number of important algebraic properties. In Section 4 we present an imperative detection algorithm, and prove that it is consistent with the algebra semantics. The algorithm is also analysed with respect to time and memory complexity. Section 5 surveys related work, and Section 6 concludes the paper.

2 The Algebra

For simplicity, we assume a discrete time model and thus let the temporal domain (denoted $\mathcal{T}$) be the set of natural numbers. 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. The simple event types from which more complex patterns are constructed, are represented by a finite set $\mathcal{P}$ of identifiers.

Definition 1 *If $A \in \mathcal{P}$, then A is a primitive event expression. If A and B are event expressions (primitive or composite), and $\tau \in \mathcal{T}$, then $A \vee B$, $A + B$, $A - B$, $A ; B$ and A_τ are composite event expressions.*

Informally, a *disjunction* $A \vee B$ represents that either of A and B occurs. A *conjunction* means that both events have occurred, in any order and 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 2 *As a running example, we consider a system with a button B, a pressure alarm P and a temperature alarm T, where some action should be performed when the button is pressed twice within two seconds, unless either of the alarms occurs in between. For this system we have $\mathcal{P} = \{\mathsf{B}, \mathsf{P}, \mathsf{T}\}$, and the described situation can be defined by the expression $(\mathsf{B};\mathsf{B})_2 - (\mathsf{P} \vee \mathsf{T})$ in the algebra.*

Occurrences are represented by *event instances*. Since the information associated with an occurrence varies between different applications, we define an underlying abstract framework rather than providing a concrete representation. Primitive event occurrences are instantaneous and atomic, but composite occurrences are associated with time intervals rather than single time points. This is necessary to achieve some of the desired algebraic properties. The interval of an event instance e is captured by the functions $\mathrm{start}(e)$ and $\mathrm{end}(e)$, where $\mathrm{end}(e)$ corresponds to the time of occurrence, and the full interval from $\mathrm{start}(e)$ to $\mathrm{end}(e)$ represents the smallest interval containing everything that caused the occurrence. The framework also contains an operator $\oplus$ by which composite instances can be constructed. E.g., each instance of $A;B$ will be constructed from one instance of A and one instance of B.

Definition 3 *An instance framework consists of:*

- *a domain D of event instances;*
- *a commutative and associative constructor function $\oplus : D \times D \to D$;*

- *a function* $\mathrm{start} : D \to \mathcal{T}$ *such that* $\mathrm{start}(e \oplus e') = \min(\mathrm{start}(e), \mathrm{start}(e'))$ *for any* $e, e' \in D$*; and*
- *a function* $\mathrm{end} : D \to \mathcal{T}$ *such that* $\mathrm{end}(e \oplus e') = \max(\mathrm{end}(e), \mathrm{end}(e'))$ *for any* $e, e' \in D$*.*

Example 4 *For systems where no additional information is associated with event occurrences, event instances can simply be represented as start and end time tuples. This would correspond to an instance framework where:*

- $D = \{\langle \tau_s, \tau_e \rangle \mid \tau_s, \tau_e \in \mathcal{T}\}$*;*
- $\langle \tau_s, \tau_e \rangle \oplus \langle \tau'_s, \tau'_e \rangle = \langle \min(\tau_s, \tau'_s), \max(\tau_e, \tau'_e) \rangle$*;*
- $\mathrm{start}(\langle \tau_s, \tau_e \rangle) = \tau_s$*; and*
- $\mathrm{end}(\langle \tau_s, \tau_e \rangle) = \tau_e$*.*

Example 5 *In some applications it is useful to tag each occurrence with additional information, e.g., to be used in the responding action. For* $A \in \mathcal{P}$*, we let* $\mathrm{dom}(A)$ *denote the domain of values associated with occurrences of* A*, and define the following instance framework:*

- D *is the powerset of* $\{\langle p, \upsilon, \tau \rangle \mid p \in \mathcal{P},\ \upsilon \in \mathrm{dom}(p),\ \tau \in \mathcal{T}\}$*;*
- $e \oplus e' = e \cup e'$*;*
- $\mathrm{start}(e) = \min(\{\tau \mid \langle p, \upsilon, \tau \rangle \in e\})$*; and*
- $\mathrm{end}(e) = \max(\{\tau \mid \langle p, \upsilon, \tau \rangle \in e\})$*.*

In our example system, the temperature alarm occurrences might carry the measured temperature value, while the pressure alarm is less sensitive and only indicates whether the pressure is too high or too low. The button instances carry no additional information, which is represented by a dummy value $\perp$ *in the framework. This corresponds to* $\mathrm{dom}(\mathsf{T}) = \mathbb{R}$*,* $\mathrm{dom}(\mathsf{P}) = \{\mathrm{high}, \mathrm{low}\}$ *and* $\mathrm{dom}(\mathsf{B}) = \{\perp\}$*. Then* $\{\langle \mathsf{T}, 38.5, 6 \rangle\}$*,* $\{\langle \mathsf{P}, \mathrm{low}, 4 \rangle\}$ *and* $\{\langle \mathsf{P}, \mathrm{low}, 4 \rangle,\ \langle \mathsf{B}, \perp, 6 \rangle\}$ *are three examples of event instances in this framework.*

For some applications, it might be more convenient to use a construction operator that does not satisfy the commutativity and associativity requirements. Most results in this article hold for such frameworks as well (see Remark 26 on page 12).

Together, all occurrences of a certain event (primitive or composite) form an *event stream.* We require that primitive event occurrences are instantaneous, and that the occurrences of each primitive event are separated in time, although two different primitive events can occur simultaneously.

Definition 6 *An event stream is a set of event instances. A primitive event stream is an event stream* S *for which the following holds:*

1. $\forall e\ (e \in S \Rightarrow \mathrm{start}(e) = \mathrm{end}(e))$

2. $\forall e\ \forall e'\ (e \in S \wedge e' \in S \wedge \mathrm{end}(e) = \mathrm{end}(e') \ \Rightarrow\ e = e')$

An *interpretation* represents a particular scenario, as it captures one of the possible ways in which the primitive events can occur.

Definition 7 *An interpretation is a function $\mathcal{I}$ mapping each identifier in $\mathcal{P}$ to a primitive event stream.*

Example 8 *Using the framework from Example 4, the following interpretation corresponds to a particular scenario with two occurrences of* T *and one occurrence each of* P *and* B*:*

$\mathcal{I}(\mathsf{B}) = \{\langle 6, 6\rangle\} \qquad \mathcal{I}(\mathsf{P}) = \{\langle 4, 4\rangle\} \qquad \mathcal{I}(\mathsf{T}) = \{\langle 1, 1\rangle,\ \langle 6, 6\rangle\}$

In the more detailed framework of Example 5, the same scenario might be represented as

$\mathcal{I}(\mathsf{B}) = \{\{\langle \mathsf{B}, \bot, 6\rangle\}\}$
$\mathcal{I}(\mathsf{P}) = \{\{\langle \mathsf{P}, \mathrm{low}, 4\rangle\}\}$
$\mathcal{I}(\mathsf{T}) = \{\{\langle \mathsf{T}, 38.2, 1\rangle\},\ \{\langle \mathsf{T}, 38.5, 6\rangle\}\}$

The naming convention is to use S, T and U for event streams, and A, B and C for event expressions. Lower case letters are used for event instances. In general, we use s for instances of an event stream S, and a for instances of the event stream defined by an event expression A, etc.

2.1 Semantics

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

Definition 9 *For event streams S and T, and for $\tau \in \mathcal{T}$, define:*

$$\begin{aligned}
\mathrm{dis}(S,T) &= S \cup T \\
\mathrm{con}(S,T) &= \{s \oplus t \mid s \in S \wedge t \in T\} \\
\mathrm{neg}(S,T) &= \{s \mid s \in S \wedge \neg\exists t(t \in T \wedge \mathrm{start}(s) \le \mathrm{start}(t) \wedge \mathrm{end}(t) \le \mathrm{end}(s))\} \\
\mathrm{seq}(S,T) &= \{s \oplus t \mid s \in S \wedge t \in T \wedge \mathrm{end}(s) < \mathrm{start}(t)\} \\
\mathrm{tim}(S,\tau) &= \{s \mid s \in S \wedge \mathrm{end}(s) - \mathrm{start}(s) \le \tau\}
\end{aligned}$$

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

Definition 10 *The meaning of an event expression for a given interpretation $\mathcal{I}$ is defined as follows:*

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

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.

Example 11 *Let $\mathcal{I}$ be the interpretation defined in Example 8. This scenario gives the following result, for the simple framework and for the framework with values, from Examples 4 and 5, respectively:*

Simple framework	*Framework with values*
$[\![\mathsf{B}\vee\mathsf{P}]\!]^{\mathcal{I}}=\{\langle 4,4\rangle, \langle 6,6\rangle\}$	$[\![\mathsf{B}\vee\mathsf{P}]\!]^{\mathcal{I}}=\{\{\langle \mathsf{P},\text{low},4\rangle\}, \{\langle \mathsf{B},\bot,6\rangle\}\}$
$[\![\mathsf{P}+\mathsf{T}]\!]^{\mathcal{I}}=\{\langle 1,4\rangle, \langle 4,6\rangle\}$	$[\![\mathsf{P}+\mathsf{T}]\!]^{\mathcal{I}}=\{\{\langle \mathsf{P},\text{low},4\rangle,\langle \mathsf{T},38.2,1\rangle\}, \{\langle \mathsf{P},\text{low},4\rangle,\langle \mathsf{T},38.5,6\rangle\}\}$
$[\![\mathsf{T};\mathsf{B}]\!]^{\mathcal{I}}=\{\langle 1,6\rangle\}$	$[\![\mathsf{T};\mathsf{B}]\!]^{\mathcal{I}}=\{\{\langle \mathsf{T},38.2,1\rangle,\langle \mathsf{B},\bot,6\rangle\}\}$
$[\![(\mathsf{P}+\mathsf{T})-\mathsf{B}]\!]^{\mathcal{I}}=\{\langle 1,4\rangle\}$	$[\![(\mathsf{P}+\mathsf{T})-\mathsf{B}]\!]^{\mathcal{I}}=\{\{\langle \mathsf{P},\text{low},4\rangle,\langle \mathsf{T},38.2,1\rangle\}\}$
$[\![(\mathsf{P}+\mathsf{T})_2]\!]^{\mathcal{I}}=\{\langle 4,6\rangle\}$	$[\![(\mathsf{P}+\mathsf{T})_2]\!]^{\mathcal{I}}=\{\{\langle \mathsf{P},\text{low},4\rangle,\langle \mathsf{T},38.5,6\rangle\}\}$

Figure 1 presents this scenario graphically.

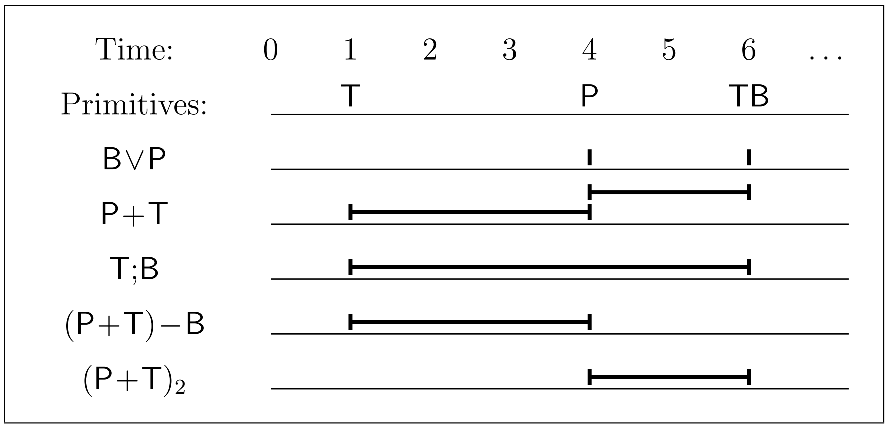


Fig. 1. Graphical representation of Example 11.

These definitions result in an algebra with simple semantics and intuitive algebraic properties, but which cannot be implemented efficiently. In particular, sequence and conjunction result in many simultaneous occurrences, and detecting all of them correctly requires that all occurrences of some constituent events are stored throughout the system lifetime.

Example 12 *Figure 2 shows the detection of the expression* T+P. *Whenever there is an occurrence of* T *it should be combined with all previous occurrences of* P *to create instances of* T+P, *and vice versa. Thus, each occurrence of* T *and* P *must be stored for future use.*

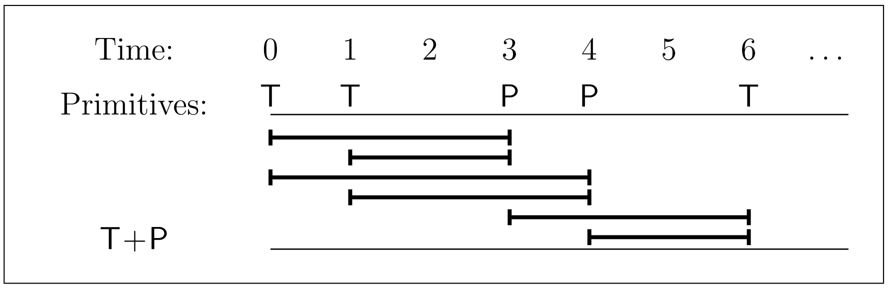


Fig. 2. Detection of T+P.

To deal with resource limitations, we introduce a formal restriction policy that defines a subset of instances that must be detected. The basic idea is to ignore simultaneous occurrences, while at the same time retaining the desired properties of the semantics.

The restriction policy is defined as a binary relation *rem* over event streams, 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. Rather than computing $[\![A]\!]$ for a given event expression A, an implementation of the algebra should compute an event stream S' for which $rem([\![A]\!], S')$ holds.

Definition 13 *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. $\forall s'_1, s'_2 \ ((s'_1 \in S' \wedge s'_2 \in S' \wedge \mathrm{end}(s'_1) = \mathrm{end}(s'_2)) \ \Rightarrow \ s'_1 = s'_2)$

Example 14 *Figure 3 shows the detected instances of* (T+P);B *in a particular scenario, and two valid restrictions* S'_1 *and* S'_2, *(i.e., both* $rem([\![(\mathsf{T}+\mathsf{P});\mathsf{B}]\!], S'_1)$ *and* $rem([\![(\mathsf{T}+\mathsf{P});\mathsf{B}]\!], S'_2)$ *hold). To see this, consider first the two instances with end time* 4. *The third criterion in the definition of rem demands that only one of them is included in the restricted stream. The first and second criteria states that one of them must be included, and that we must in fact select the one that starts at time* 2. *In the same way, from the three instances with end time* 6 *we must include exactly one in the restricted stream, and it must be one of the two with start time* 2. *The choice between them, however,* *is arbitrary, and thus there are two valid restrictions, S'_1 and S'_2.*

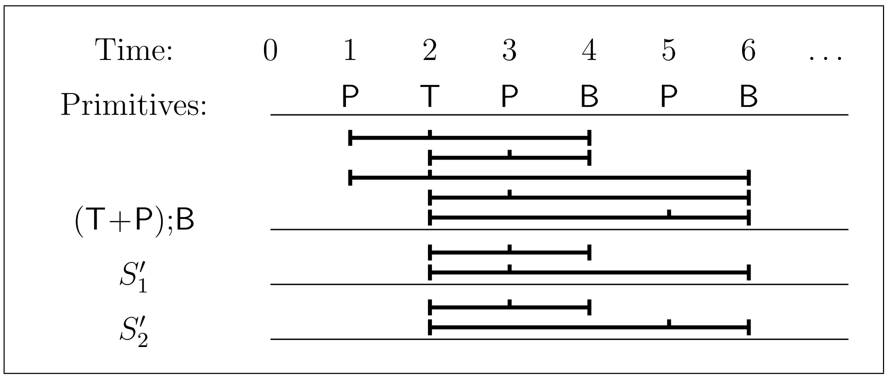


Fig. 3. Detection of (T+P);B, and the two valid restrictions S'_1 and S'_2.

For the user of the algebra, a significant property of this policy is that at any time when there are one or more occurrences of A according to the semantics defined above, one of them will be detected (as ensured by the second criterion).

The fact that it is always an instance with maximum start time that is detected is probably less significant to the user. However, this choice is crucial to achieve the desired efficiency since it allows the restriction policy to be applied recursively to all subexpressions, without affecting the overall result.

Applying restriction at all levels of nesting would normally require a user of the algebra to understand how the restrictions of different subexpressions interfere with each other, and their effect on different operator combinations. To avoid this, the restriction policy has been designed in such a way that applying it to all subexpressions gives a result which is consistent with applying it only at the top level. This property is formalised by Theorem 15 below. As a result, from the point of view of a user, the restriction policy is applied only once to the whole expression, but an implementation can freely apply it to the subexpressions as well.

Theorem 15 *If $rem(S, S')$ and $rem(T, T')$ hold, than for any event stream U and $\tau \in \mathcal{T}$ the following implications hold:*

- $rem(\text{dis}(S', T'),\ U) \quad \Rightarrow \quad rem(\text{dis}(S, T),\ U)$
- $rem(\text{con}(S', T'),\ U) \quad \Rightarrow \quad rem(\text{con}(S, T),\ U)$
- $rem(\text{neg}(S', T'),\ U) \quad \Rightarrow \quad rem(\text{neg}(S, T),\ U)$
- $rem(\text{seq}(S', T'),\ U) \quad \Rightarrow \quad rem(\text{seq}(S, T),\ U)$
- $rem(\text{tim}(S', \tau),\ U) \quad \Rightarrow \quad rem(\text{tim}(S, \tau),\ U)$

PROOF. The proof can be found in Appendix A. □

Although a single stream may have several valid restrictions, they all share an important characteristic: They are equivalent with respect to instance start and end times.

Proposition 16 *If rem(S,T) and rem(S,T') then for each $t \in T$ there exists a $t' \in T'$ with* $\text{start}(t) = \text{start}(t')$ *and* $\text{end}(t) = \text{end}(t')$.

PROOF. Since $T \subseteq S$, $t \in S$. By the second condition in the definition of *rem*, there exists some $t' \in T'$ such that $\text{start}(t) \leq \text{start}(t')$ and $\text{end}(t) = \text{end}(t')$. We also have $t' \in S$, and thus there is some $t'' \in T$ such that $\text{start}(t') \leq \text{start}(t'')$ and $\text{end}(t') = \text{end}(t'')$. According to the third condition in the definition of *rem* this implies $t = t''$, which means that we have $\text{start}(t) \leq \text{start}(t') \leq \text{start}(t)$ and thus $\text{start}(t') = \text{start}(t)$. □

3 Properties

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

Definition 17 *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 proposition shows that it satisfies the substitutive condition, and hence defines structural congruence over event expressions.

Proposition 18 *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 directly from Definition 10. □

The laws presented later in this section identify expressions that are semantically equivalent with respect to the operator semantics, but in order to deal with resource limitations, we expect an implementation of the algebra to compute an event stream S such that *rem*$([\![A]\!], S)$, rather than the full $[\![A]\!]$. Since *rem* is a predicate and not a function, detecting A might potentially yield a different stream than detecting A', even when $A \equiv A'$. Consequently, it should be clarified to what extent restriction policy affects expression equivalence.

Proposition 19 *If $A \equiv A'$ and rem($[\![A]\!], S$), then rem($[\![A']\!], S$).*

PROOF. Since $A \equiv A'$ implies that $[\![A]\!] = [\![A']\!]$, this holds trivially. □

Thus, $A \equiv A'$ ensures that for any implementation consistent with the restriction policy, the detected occurrences of A is always a valid result for A' as well. Any reasoning based on the algebra semantics and the restriction policy, and not on the details of a particular detection algorithm, will be equally valid for equivalent expressions.

The next proposition ensures that although the detection of A and A' may not be exactly identical, they must be equivalent with respect to time.

Proposition 20 *If $A \equiv A'$, rem($[\![A]\!], S$) and rem($[\![A']\!], S'$), then for any $s \in S$ there exists a $s' \in S'$ with $\mathrm{start}(s) = \mathrm{start}(s')$ and $\mathrm{end}(s) = \mathrm{end}(s')$.*

PROOF. This follows straightforwardly from Proposition 16. □

The algebraic properties are given in the following theorems. Derived laws are indicated by an asterisk ($*$), and the proofs can be found in Appendix B.

Theorem 21 *For event expressions A, B and C, the following laws hold:*

1.	$A \vee A \equiv A$	6.	$A;(B;C) \equiv (A;B);C$
2.	$A \vee B \equiv B \vee A$	7.	$(A \vee B)+C \equiv (A+C) \vee (B+C)$
3.	$A+B \equiv B+A$	$*$8.	$A+(B \vee C) \equiv (A+B) \vee (A+C)$
4.	$A \vee (B \vee C) \equiv (A \vee B) \vee C$	9.	$(A \vee B);C \equiv (A;C) \vee (B;C)$
5.	$A+(B+C) \equiv (A+B)+C$	10.	$A;(B \vee C) \equiv (A;B) \vee (A;C)$

Theorem 22 *For event expressions A, B and C, the following laws hold:*

11.	$(A \vee B)-C \equiv (A-C) \vee (B-C)$	$*$15.	$(A-B)-B \equiv A-B$
12.	$(A+B)-C \equiv ((A-C)+B)-C$	$*$16.	$(A-B)-C \equiv (A-C)-B$
$*$13.	$(A+B)-C \equiv (A+(B-C))-C$	17.	$(A;B)-C \equiv ((A-C);B)-C$
14.	$(A-B)-C \equiv A-(B \vee C)$	18.	$(A;B)-C \equiv (A;(B-C))-C$

Theorem 23 *For event expressions A and B, and $\tau \in \mathcal{T}$, the following laws hold:*

19.	$(A{\vee}B)_\tau \equiv A_\tau{\vee}B_\tau$	24.	$(A;B)_\tau \equiv (A_\tau;B)_\tau$
20.	$(A{+}B)_\tau \equiv (A_\tau{+}B)_\tau$	25.	$(A;B)_\tau \equiv (A;B_\tau)_\tau$
∗21.	$(A{+}B)_\tau \equiv (A{+}B_\tau)_\tau$	26.	$A \equiv A_\tau \quad \textit{if } A \in \mathcal{P}$
22.	$(A{-}B)_\tau \equiv A_\tau{-}B$	27.	$(A_\tau)_{\tau'} \equiv A_{\min(\tau,\tau')}$
23.	$(A{-}B)_\tau \equiv (A{-}B_\tau)_\tau$	∗28.	$(A_\tau)_{\tau'} \equiv (A_{\tau'})_\tau$

Finally, we introduce the notion of an empty event that never occurs, and laws related to this.

Definition 24 *Let the constant* 0 *denote the empty event, semantically defined as* $[\![0]\!]^{\mathcal{I}} = \emptyset$ *for any interpretation* $\mathcal{I}$.

Theorem 25 *For an event expression* A *the following laws hold:*

29.	$0{\vee}A \equiv A$	34.	$0{-}A \equiv 0$
∗30.	$A{\vee}0 \equiv A$	35.	$A{-}0 \equiv A$
31.	$0{+}A \equiv 0$	36.	$0;A \equiv 0$
∗32.	$A{+}0 \equiv 0$	37.	$A;0 \equiv 0$
33.	$A{-}A \equiv 0$	38.	$0_\tau \equiv 0$

PROOF. These laws follow straightforwardly from the operator semantics and the definition of 0. □

Alternatively, 0 can be defined as shorthand for an expression $A{-}A$, where A is an arbitrary event expression (compare with law 33).

Remark 26 *For instance frameworks with a construction operator that does not satisfy the commutativity and associativity requirements, all laws except number 3 (requires commutativity), 5 and 6 (require associativity) still hold. Note that the laws derived from these three laws (8, 13, 21, 30 and 32) hold anyway, since they can be proven individually.*

4 Detection algorithm

In this section, we present an imperative algorithm that, for a given event expression E and interpretation $\mathcal{I}$, computes an event stream S for which $\mathit{rem}([\![E]\!]^{\mathcal{I}}, S)$ 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 according to a postorder traversal of the expression[1], and we let E^i

[1] In fact, any ordering where a subexpression is given a higher number than its constituents would be acceptable.

denote subexpression number i. Consequently, we always have $E^m = E$ and $E^1 \in \mathcal{P}$. For example, with $E = (\mathsf{T}+\mathsf{P})-\mathsf{B}$, we have $E^1 = \mathsf{T}$, $E^2 = \mathsf{P}$, $E^3 = (\mathsf{T}+\mathsf{P})$, $E^4 = \mathsf{B}$, and $E^5 = E$.

The algorithm is given in Figure 5. It is executed once every time tick, and computes the current instance of E from the current instances of the primitive events, and from stored information about the past. The main loop from 1 to m corresponds to a postorder traversal of E. 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 variables used in the algorithm can be divided into three categories (see Figure 4). *Persistent* variables store information that must be remembered from one time tick to the next in order to detect the event properly. Since each subexpression requires its own persistent variables, they are indexed from 1 to m. For conjunction, variables l_i and r_i are used to store significant past instances of the left and right subexpression, respectively. The l_i variable is also used in a similar way for sequence, together with Q_i which holds a set of other significant past instances of the left subexpression. For negation, t_i stores the latest start time of the past instances of the left subexpression.

Auxiliary variables are indexed in the same way as the persistent variables, but pass information from a subexpression to its parent within a tick. In particular, a_i is used to store the current instance of E^i. The S_i variables are one of the keys to ensuring that there are static memory bounds for the algorithm, and their role is discussed below.

Finally, there are *temporary* variables that are used locally within a single subexpression. Since these are not intended to store values until next tick, nor between subexpressions, they can be freely shared and are not indexed.

Category	**Variable**	**Type**	**Initial value**
Persistent	l_i, r_i	instance	$\langle\rangle$
	Q_i	instance set	$\emptyset$
	t_i	time	-1
Auxiliary	a_i	instance	
	S_i	time set	$\emptyset$
Temporary	t	time	
	e, e'	instance	
	Q'	instance set	

Fig. 4. Variables used in the detection algorithm.

In order to comply with the restriction policy, the parts of the algorithm responsible for disjunction and conjunction have to ensure that when choosing between two simultaneous occurrences, the choice resulting in the latest start time is taken.

```
for $i$ from $1$ to $m$
    if $E^i \in \mathcal{P}$ then
        if there is a current instance $e$ of $E^i$ then $a_i := e$
        else $a_i := \langle\rangle$
    if $E^i = E^j \vee E^k$ then
        if $\mathrm{start}(a_j) \leq \mathrm{start}(a_k)$ then $a_i := a_k$ else $a_i := a_j$
        $S_i := S_j \cup S_k$
    if $E^i = E^j + E^k$ then
        if $\mathrm{start}(l_i) < \mathrm{start}(a_j)$ then $l_i := a_j$
        if $\mathrm{start}(r_i) < \mathrm{start}(a_k)$ then $r_i := a_k$
        if $l_i = \langle\rangle \vee r_i = \langle\rangle \vee (a_j = \langle\rangle \wedge a_k = \langle\rangle)$ then $a_i := \langle\rangle$
        else if $\mathrm{start}(a_k) \leq \mathrm{start}(a_j)$ then $a_i := a_j \oplus r_i$
            else $a_i := l_i \oplus a_k$
        $S_i := S_j \cup S_k \cup \{\mathrm{start}(l_i),\ \mathrm{start}(r_i)\} \backslash \{-1\}$
    if $E^i = E^j - E^k$ then
        if $t_i < \mathrm{start}(a_k)$ then $t_i := \mathrm{start}(a_k)$
        if $t_i < \mathrm{start}(a_j)$ then $a_i := a_j$ else $a_i := \langle\rangle$
        $S_i := S_j$
    if $E^i = E^j ; E^k$ then
        $e' := \langle\rangle$
        foreach $e$ in $Q_i \cup \{l_i\}$
            if $\mathrm{end}(e) < \mathrm{start}(a_k) \wedge \mathrm{start}(e') < \mathrm{start}(e)$ then $e' := e$
        if $e' \neq \langle\rangle$ then $a_i := a_k \oplus e'$ else $a_i := \langle\rangle$
        $Q' := \emptyset$
        foreach $t$ in $S_k$
            $e' := \langle\rangle$
            foreach $e$ in $Q_i \cup \{l_i\}$
                if $\mathrm{end}(e) < t \wedge \mathrm{start}(e') < \mathrm{start}(e)$ then $e' := e$
            $Q' := Q' \cup \{e'\}$
        $Q_i := Q'$
        if $\mathrm{start}(l_i) < \mathrm{start}(a_j)$ then $l_i := a_j$
        $S_i := S_j \cup \{\mathrm{start}(e) \mid e \in Q_i \cup \{l_i\}\} \backslash \{-1\}$
    if $E^i = (E^j)_\tau$ then
        if $\mathrm{end}(a_j) - \mathrm{start}(a_j) \leq \tau$ then $a_i := a_j$ else $a_i := \langle\rangle$
        $S_i := S_j$
```

Fig. 5. The detection algorithm. For an event expression E, the content of a_m at the end of each time tick form an event stream $\mathcal{A}(m)$ which satisfies $\mathit{rem}([\![E]\!], \mathcal{A}(m))$. Initially, $t_i = -1$, $l_i = r_i = \langle\rangle$ and $S_i = Q_i = \emptyset$ for $1 \leq i \leq m$.

Even with the restriction policy, the problem remains in the case of a sequence $E^j;E^k$ to know what instances of E^j that will be the best match for future E^k instances. Since non-overlapping is required by the sequence operator, it is not enough to store the instance of E^j with latest end time so far. In order to achieve bounded memory, however, the number of E^j instances to store in Q_i must be bounded. Our solution to this problem is to propagate not only full detections of E^k but also information about possible start times of future E^k instances, i.e., the start times of partial detections. This information, represented by the S_i variables, is collected from all subexpression. Fortunately, the number of simultaneously active "possible start times" can be bounded, which allows a bounded memory implementation of the algebra.

After executing the algorithm, the variable a_i contains the detected occurrence of E^i in the current tick, or $\langle\rangle$ if there is none. To connect this to the algorithm semantics, we define an event stream corresponding to each a_i variable.

Definition 27 *For $1 \leq i \leq m$, define*

$$\mathcal{A}(i) = \{e \mid e \text{ is the value of } a_i \text{ at the end of some time tick} \wedge e \neq \langle\rangle\}$$

Thus, the output of the algorithm is the event stream $\mathcal{A}(m)$, and as will be established in the next section (by Theorem 36), this event stream satisfies $\mathit{rem}([\![E]\!], \mathcal{A}(m))$.

4.1 Algorithm Correctness

In order to prove that this algorithm correctly implements the algebra semantics and the restriction policy, we first introduce a number of predicates that capture different correctness properties of the algorithm. We proceed by proving the correctness of a single operator at a single time tick, for each of these correctness properties. The full correctness proof is organised as two nested inductions: an inner induction over the subexpressions of E, and an outer induction over time.

4.1.1 Correctness properties

The fact that the output of the algorithm at a single time tick is consistent with the restriction policy, is captured by what can be thought of as a pointwise restriction predicate, and a lemma that relates it to the ordinary restriction policy.

Definition 28 *For an event instance* e*, an event stream* S *and* $\tau \in \mathcal{T}$*, define* $valid(e, S, \tau)$ *to hold if:*

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

Lemma 29 *For an event stream* S *and event instances* $e_0, e_1, e_2, \ldots$ *such that* $valid(e_\tau, S, \tau)$ *holds for any* $\tau \in \mathcal{T}$*, let* $S' = \{e_0, e_1, e_2, \ldots\} \backslash \{\langle\rangle\}$*. 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. $\square$

The following property represents that the detected instance of E^i is correct with respect to the instances detected by the subexpressions.

Definition 30 *Define* $acorr(i, \tau)$ *as follows:*

- *For* $E^i \in \mathcal{P}$*,* $acorr(i, \tau)$ *holds iff* $valid(a_i, [\![E^i]\!], \tau)$
- *For* $E^i = E^j \vee E^k$*,* $acorr(i, \tau)$ *holds iff* $valid(a_i, \mathrm{dis}(\mathcal{A}(j), \mathcal{A}(k)), \tau)$
- *For* $E^i = E^j + E^k$*,* $acorr(i, \tau)$ *holds iff* $valid(a_i, \mathrm{con}(\mathcal{A}(j), \mathcal{A}(k)), \tau)$
- *For* $E^i = E^j - E^k$*,* $acorr(i, \tau)$ *holds iff* $valid(a_i, \mathrm{neg}(\mathcal{A}(j), \mathcal{A}(k)), \tau)$
- *For* $E^i = E^j ; E^k$*,* $acorr(i, \tau)$ *holds iff* $valid(a_i, \mathrm{seq}(\mathcal{A}(j), \mathcal{A}(k)), \tau)$
- *For* $E^i = E^j_{\tau'}$*,* $acorr(i, \tau)$ *holds iff* $valid(a_i, \mathrm{tim}(\mathcal{A}(j), \tau'), \tau)$

To achieve bounded memory, the sequence operator requires some knowledge about what is stored in the persistent variables of its subexpressions. This information is propagated by the S_i variables, and the following predicate indirectly defines their correctness. Informally, it states that the start time of any detected non-instantaneous event was already propagated in the previous tick, and that the S_i variables are not updated with arbitrary values, only with the current time.

Definition 31 *Define* $pcorr(i, \tau)$ *to hold iff the following criteria hold:*

1. $a_i = \langle\rangle \;\vee\; \mathrm{start}(a_i) = \tau \;\vee\; \mathrm{start}(a_i) \in S$
2. $\forall t \; (t \in S_i \Rightarrow (t = \tau \vee t \in S))$

where S *was the content of* S_i *at the start of the current time tick.*

The operators that require information about what has happened in the past, store this state information in the persistent variables r_i, l_i, t_i and Q_i. The following predicate defines what they should contain at the start of tick τ.

Definition 32 *Define state(i, τ) as follows:*

- *For $E^i \in \mathcal{P}$, $E^i = E^j \vee E^k$ and $E^i = E^j_{\tau'}$, state(i, τ) holds trivially.*
- *For $E^i = E^j + E^k$, state(i, τ) holds iff*
  - *l_i is an element in $\{e \mid e \in \mathcal{A}(j) \wedge \text{end}(e) < \tau\} \cup \{\langle\rangle\}$ with maximum start time; and*
  - *r_i is an element in $\{e \mid e \in \mathcal{A}(k) \wedge \text{end}(e) < \tau\} \cup \{\langle\rangle\}$ with maximum start time.*
- *For $E^i = E^j - E^k$, state(i, τ) holds iff*
  - *t_i is the maximum element in $\{\text{start}(e) \mid e \in \mathcal{A}(k) \wedge \text{end}(e) < \tau\} \cup \{-1\}$.*
- *For $E^i = E^j ; E^k$, state(i, τ) holds iff*
  - *l_i is an element in $\{e \mid e \in \mathcal{A}(j) \wedge \text{end}(e) < \tau\} \cup \{\langle\rangle\}$ with maximum start time; and*
  - *for each $t \in S_k$ such that $\{e \mid e \in \mathcal{A}(j) \wedge \text{end}(e) < t\}$ is non-empty, Q_i contains an element with maximum start time from that set.*

4.1.2 Correctness results

Focusing first on the result of a single subexpression at a single time tick, we show that each of the three correctness properties hold under some given assumptions.

Lemma 33 *Assume that state(i, τ) held at the start of the current tick and that pcorr(n, τ) and acorr(n, τ) hold for all $1 \leq n < i$. Then state$(i, \tau + 1)$, pcorr(i, τ) and acorr(i, τ) hold after executing the loop body once.*

PROOF. The proof can be found in Appendix C. □

The correctness lemma above is used in the inductive step of the two nested induction proofs over the expression and over time, respectively.

Lemma 34 (Inner induction) *Let τ be the current time, and assume that for each $1 \leq i \leq m$ state(i, τ) held at the start of this tick. Then state$(i, \tau+1)$ and acorr(i, τ) holds for each $1 \leq i \leq m$ after executing the whole detection algorithm.*

PROOF. In addition to to the assumption about *state*, assume that after executing the loop body $n - 1$ times, *pcorr*(i, τ) and *acorr*(i, τ) hold for all

$1 \leq i < n$. As a base case, this clearly holds for $n = 1$. Then $\mathit{state}(n, \tau{+}1)$, $\mathit{pcorr}(n, \tau)$ and $\mathit{acorr}(n, \tau)$ hold after loop iteration n, according to Lemma 33. By induction, the lemma holds. □

Lemma 35 (Outer induction) *For any i such that $1 \leq i \leq m$, and any $\tau \in \mathcal{T}$ $\mathit{acorr}(i, \tau)$ holds after executing the algorithm at ticks 0 to τ.*

PROOF. For the base case we see that $\mathit{state}(i, 0)$ holds in an initial state where $t_i = -1$, $l_i = r_i = \langle\rangle$ and $Q_i = \emptyset$. For the inductive case: Assume that for some $\tau \in \mathcal{T}$, $\mathit{state}(i, \tau)$ holds at the start of tick τ. Then, according to Lemma 34, $\mathit{state}(i, \tau{+}1)$ and $\mathit{acorr}(i, \tau)$ holds after execution the algorithm, and thus $\mathit{state}(i, \tau{+}1)$ holds at the start of tick $\tau{+}1$. By induction over time the lemma thus holds for any $\tau \in \mathcal{T}$. □

So far, we have only shown that the result produced for E^i is correct with respect to the result produced by its subexpressions. Now, we take the final step and prove the correctness of the algorithm in the following theorem.

Theorem 36 *For any i such that $1 \leq i \leq m$, $\mathit{rem}(\llbracket E^i \rrbracket, \mathcal{A}(i))$ holds.*

PROOF. Assume that for some i, $\mathit{rem}(\llbracket E^n \rrbracket, \mathcal{A}(n))$ holds for all $1 \leq n < i$. For the base case, this trivially holds for $i = 1$. According to Lemma 35, $\mathit{acorr}(i, \tau)$ holds at the end of tick τ. For $E^i \in \mathcal{P}$, we know from the definition of acorr that $\mathit{valid}(a_i, \llbracket E^i \rrbracket, \tau)$ holds at the end of tick τ, and then Lemma 29 ensures $\mathit{rem}(\llbracket E^i \rrbracket, \mathcal{A}(i))$. If $E^i = E^j \vee E^k$, the definition of acorr implies that $\mathit{valid}(a_i, \mathrm{dis}(\mathcal{A}(j), \mathcal{A}(k)), \tau)$ holds at the end of tick τ, so by Lemma 29 we have $\mathit{rem}(\mathrm{dis}(\mathcal{A}(j), \mathcal{A}(k)), \mathcal{A}(i))$. According to Theorem 15, this and the induction assumption that $\mathit{rem}(\llbracket E^j \rrbracket, \mathcal{A}(j))$ and $\mathit{rem}(\llbracket E^k \rrbracket, \mathcal{A}(k))$ hold (since $j < i$ and $k < i$), implies $\mathit{rem}(\mathrm{dis}(\llbracket E^j \rrbracket, \llbracket E^k \rrbracket), \mathcal{A}(i))$ and thus $\mathit{rem}(\llbracket E^i \rrbracket, \mathcal{A}(i))$. The proofs for the remaining operators are analogous. □

4.2 Algorithm improvements

To simplify presentation and correctness analysis, the algorithm uses set variables, and a time driven execution style was assumed where the algorithm is executed once every time instant. However, these design alternatives also have an impact on the efficiency of the algorithm, which must be addressed and resolved.

Considering first the issue of time triggered execution, we can see that in time ticks where no primitive events occur, none of the persistent variables are changed by the algorithm, and the a_i variables all become $\langle\rangle$. In fact, this means that $\mathcal{A}(m)$ remains the same if the algorithm is executed only in ticks when at least one of the primitive events in E has occurred. Consequently, the algorithm presented here can be used with little or no changes also in an event driven setting where the execution of the algorithm is triggered by primitive event occurrences rather than at each tick. If primitive events are non-simultaneous and always trigger the algorithm in the same order as they occur, the algorithm can be used without changes. Otherwise, some precautions must be taken to ensure that occurrences are processed in the right order.

This improvement could be taken a step further by processing only subexpressions that are affected by the current primitive event occurrences. The identification of what parts of the tree to consider could either be done statically with respect to the primitive events, or dynamically based also on the detection result of the subexpressions. The details of this optimisation remains to be investigated, though.

Turning to the set variables, we notice that the worst part of the algorithm, from a complexity point of view, is the nested foreach constructs in the sequence part. However, this source of complexity can be avoided, without compromising the correctness of the algorithm, if the set variables S_i and Q_i are represented as ordered structures. First, note that Q_i never contain fully overlapping instances. New values that are added to Q_i always come from l_i, and whenever l_i is updated, both the start and end time of the new instance is greater than those of the previous instance. Thus, if Q_i is ordered with respect to end times, it will also be ordered with respect to start time.

The time complexity of the S_i assignments is not affected by the ordered representation. For the sequence part of the algorithm, this follows from the

```
**foreach** $t$ **in** $S_k$
  $e' := \langle\rangle$
  **foreach** $e$ **in** $Q_i \cup \{l_i\}$
    **if** $\text{end}(e) < t \wedge \text{start}(e') < \text{start}(e)$ **then**
      $e' := e$
    $Q' := Q' \cup \{e'\}$
```

```
$Q_i := Q_i \cup \{l_i\}$
$sp := \text{length}(S_k)$
$qp := \text{length}(Q_i)$
**while** $sp > 0 \wedge qp > 0$
  $t := S_k[sp]$
  $e := Q_i[qp]$
  **if** $\text{end}(e) < t$ **then**
    $Q' := Q' \cup \{e\}$
    $sp := sp - 1$
  **else** $qp := qp - 1$
```

Fig. 6. Part of the original sequence operator algorithm (left), and the improved version for the case when Q_i and S_k are ordered (right).

fact that Q_i is ordered with respect to start times. The assignment of Q_i is done by means of a temporary set variable Q' that is populated by the best match, from the instances currently stored in Q_i and l_i, for each element in S_k. In the original detection algorithm, this is performed by the two nested foreach constructs shown in Figure 6 (left), but when Q_i and S_k are ordered it can be accomplished by a single pass over the two structures together, as shown in Figure 6 (right). An array style notation is used for references to individual elements of an ordered structure (e.g., $S_k[1]$ for the first element of S_k).

It is also worth pointing out that the presented algorithm is designed for detection of arbitrary expressions, and thus 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. Also, the assignments of S_i variables can be removed for all subexpressions except those occurring within the right-hand argument of a sequence operator. A concrete example of this is given in Figure 7.

```
if there is a current instance $e$ of T then $a_1 := e$ else $a_1 := \langle\rangle$
if there is a current instance $e$ of P then $a_2 := e$ else $a_2 := \langle\rangle$
if $\text{start}(l_3) < \text{start}(a_1)$ then $l_3 := a_1$
if $\text{start}(r_3) < \text{start}(a_2)$ then $r_3 := a_2$
if $l_3 = \langle\rangle \vee r_3 = \langle\rangle \vee (a_1 = \langle\rangle \wedge a_2 = \langle\rangle)$ then $a_3 := \langle\rangle$
else if $\text{start}(a_2) \leq \text{start}(a_1)$ then $a_3 := a_1 \oplus r_3$
    else $a_3 := l_3 \oplus a_2$
if there is a current instance $e$ of B then $a_4 := e$ else $a_4 := \langle\rangle$
if $t_5 < \text{start}(a_4)$ then $t_5 := \text{start}(a_4)$
if $t_5 < \text{start}(a_3)$ then $a_5 := a_3$ else $a_5 := \langle\rangle$
```

Fig. 7. Statically simplified algorithm for detecting (T+P)−B. Initially, $t_5 = -1$ and $l_3 = r_3 = \langle\rangle$.

4.3 Complexity analysis

Most parts of the algorithm are fairly straightforward to analyse with respect to time and memory usage, but we need to establish bounds on the set variables S_i, Q_i and Q'. For this, let $|X|$ denote the maximum size of a set variable X.

Proposition 37 *If $E^i = E^j;E^k$ then $|Q_i| \leq |S_k|$, otherwise $|Q_i| = 0$. We also have $|Q'| = \max_{1 \leq i \leq m}(|Q_i|)$.*

PROOF. This follows straightforwardly from the assignments of Q_i and Q' in the algorithm. □

Proposition 38 *For any i such that $1 \leq i \leq m$, we have $|S_i| < \text{subexp}(E^i)$ where $\text{subexp}(E)$ denotes the number of subexpressions in E.*

PROOF. In the base case $i = 1$, we have $E^i \in \mathcal{P}$ and thus $|S_i| = 0$ and $\text{subexp}(E^i) = 1$, which clearly satisfies the claim. For the inductive case we assume that $|S_n| < \text{subexp}(E^n)$ holds for all $1 \leq n < i$. If $E^i \in \mathcal{P}$ we can repeat the proof for the base case. If $E^i = (E^j)_\tau$, then $|S_i| = |S_j|$, and since $j < i$, the assumption implies that $|S_j| < \text{subexp}(E^j)$. Thus, we have $|S_i| < \text{subexp}(E^j) < \text{subexp}(E^i)$. In the remaining cases where E^i is a binary operator applied to E^j and E^k, we have $j < i$ and $k < i$ and thus the assumption implies that $|S_j| \leq \text{subexp}(E^j) - 1$ and $|S_k| \leq \text{subexp}(E^k) - 1$. From the assignments of S_i in the algorithm, we see that $|S_i| \leq |S_j| + |S_k| + 2$ holds for all operators (for sequence, we use the fact that $|Q_i| \leq |S_k|$). Thus, $|S_i| \leq |S_j| + |S_k| + 2 \leq \text{subexp}(E^j) + \text{subexp}(E^k) < \text{subexp}(E^i)$, which concludes the proof. □

The memory and time complexity of the algorithm also depends on the particularities of the instance framework. Hence, we introduce the parameter ω to denote the maximum memory needed to store an instance in the current framework. An instance of a subexpression of E is constructed from at most $\lceil m/2 \rceil$ primitive instances (one from each leaf in the expression tree). Thus, assuming that primitive instances are of bounded size, and that the size of $a \oplus b$ is bounded whenever the size of a and b is, the instance size is bounded. For the time analysis, we assume that the time it takes to perform the $\oplus$ operation is proportional to ω, or lower. As previously, m denotes the number of subexpressions in E.

Theorem 39 *The memory complexity of the algorithm is $\mathrm{O}(m^2\omega)$.*

PROOF. Since $\text{subexp}(E^i) \leq m$ for any $1 \leq i \leq m$, it follows from Propositions 37 and 38 that $|Q'| \leq m$ and that $|S_i| \leq m$ and $|Q_i| \leq m$ for any $1 \leq i \leq m$. This means that the algorithm stores at most $\mathrm{O}(m^2)$ instances and time values. □

Theorem 40 *The time complexity of the algorithm is $\mathrm{O}(m^2\omega)$.*

PROOF. The algorithm performs m iterations of the main loop, each iteration executing one of the operator specific parts of the loop body. Only the code for the sequence operator contains loop structures, so for the other operators the primary source of complexity are the assignments of the set variables S_i, and they can be performed in $\mathrm{O}(|S_i|\omega)$, also when the S_i variables are ordered. For the sequence operator we assume that the improvement shown in Figure 6 (right) is applied, which means that the code has two loop structures, each with a body that runs in $\mathrm{O}(\omega)$ time. The foreach loop iterates $|Q_i| + 1$ times, and the while loop $|Q_i| + 1 + |S_k|$ times. Finally, the set assignment can be performed in $\mathrm{O}(|S_i|\omega)$ time when Q_i is ordered with respect to start time. Altogether, since Propositions 37 and 38 ensures that $|Q_i|$, $|S_i|$ and $|S_k|$ are less than or equal to m, the code for each operator can be executed in $\mathrm{O}(m\omega)$ time. Thus, the time complexity of the whole algorithm is $\mathrm{O}(m^2\omega)$. □

Example 41 *In the simple framework of Example 4, ω is a constant factor, and thus the time and memory complexity are $\mathrm{O}(m^2)$. In the framework of Example 5, the instance size is bounded by $\lceil m/2 \rceil$, and thus the memory and time complexity of the algorithm are $\mathrm{O}(m^3)$.*

4.4 Experiments

In addition to the complexity analysis, we have conducted some basic experiments to investigate the resource requirements of the algorithm in more detail. In particular, the complexity analysis regards the worst case expression, but most expressions have significantly lower resource demands, since different operator combinations contribute very differently to the overall time and memory usage.

Expressions containing m subexpressions were created randomly, with equal probability for the five operators to occur, and for each expression, the static bound on memory footprint and worst case execution time were derived. We have assumed that sets are represented in a straightforward way, with the single optimisation that the S_i variables are only used in subexpressions occuring within the right-hand argument of a sequence operator.

Each m value is represented by 10.000 random expressions, and the 95% confidence intervals for the mean values are less than 2% of the y-value for all points.

Experiment 1: For the memory analysis, we assume that the storage of a single time value requires 1 memory unit. In the simple framework, any instance requires 2 memory units, for start and end times. In the value framework, we assume that primitive instances require 3 units (time, id and value), and the size of a composite instance $a \oplus b$ is the summed size of a and b. Figure 8 shows the mean and maximum memory usage of the sample expressions, for each m value.

The experiment shows that the detection algorithm memory usage is fairly low, also for complex expressions. The detection of an average expression consisting of 51 subexpressions requires less than 250 units of memory in the simple framework and roughly the double when all primitives carry values. Over the whole experiment, the maximum value is approximately twice as high as the average for the simple framework, and four times as high in the value framework. Although the actual worst case might be significantly higher than the maximum within the samples of 10.000 expressions investigated in the experiment, this indicates that expressions with high memory demand are very rare.

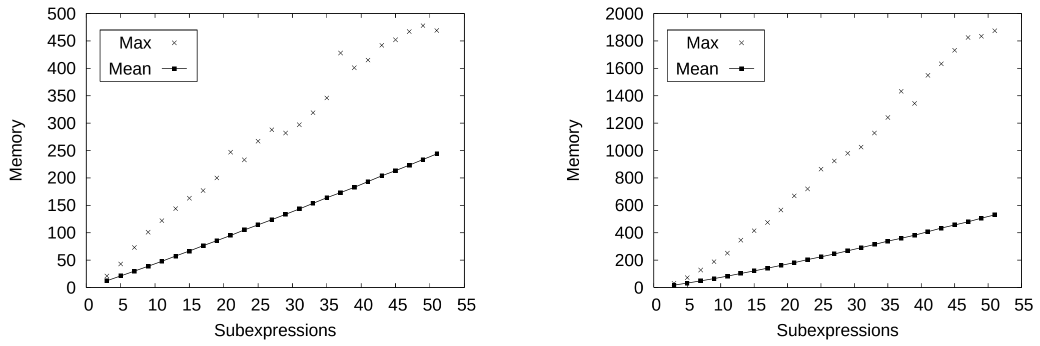
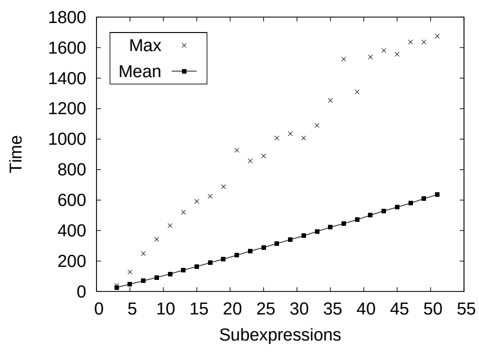


Fig. 8. Memory usage in the simple framework (left) and the value framework (right).

Experiment 2: The timing analysis is based on an abstract execution model where we assume that comparisons, arithmetic operations and assigning a time instant variable take 1 time unit. The time it takes to assign an instance variable is the same as the size of that variable, and a set assignment $S' := S$ takes $|S| * s$ time units, where s is the time it takes to assign a single element of S.

For each random expression, the worst case execution time is computed. Figure 8 shows the mean worst case execution time, as well as the maximum, for each m value. The detection of an average expression consisting of 51 subexpressions takes less than 650 time units in the simple framework, and less than 980 in the value framework. As for memory, we note that the difference between average and maximum is relatively small. For the simple framework, maximum is approximately three times higher than average, and in the value framework it is four times higher.

These experiments show that, although there exist expressions that result in fairly high resource usage, the average is significantly lower. None of the investigated expressions have memory footprint or execution time values that would prevent them from being used in an embedded setting. It is also worth pointing out that the second experiment regards worst case execution time only. The execution time of an average tick depends on the actual occurrence frequencies of the primitive events.



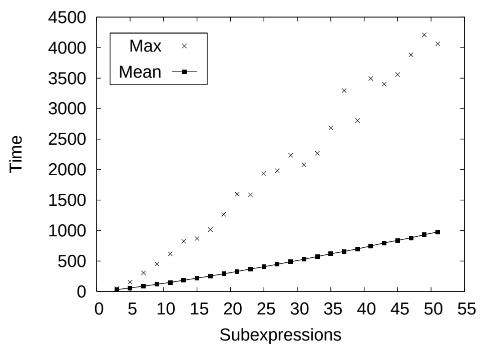


Fig. 9. Worst case execution times in the simple framework (left) and the value framework (right).

5 Related work

The operators of our algebra are influenced by work in the area of active databases, such as Snoop [9], Ode [11] and SAMOS [12]. These systems have similar event specification languages, although the underlying implementation mechanisms are based on event graphs, Petri nets and state automata, respectively. Resource efficiency is typically not the main concern in an active database, and the event specification languages are primarily designed to provide sufficient expressiveness, rather than to ensure bounded resources in the general case.

Liu et al. [20] describe how composite events can be expressed as timing constraints in Real Time Logic, and thus 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. The resulting event specification language is more expressive than ours, in that it can refer to individual instances and not just event types, e.g., *the fourth instance of A occurs before the second instance of B*, but it provides no assistance in terms of algebraic properties.

As discussed in the introduction, these and many other techniques are based on single point semantics (sometimes called *detection semantics*). It has been shown by Galton and Augusto [15] that this gives an unintended meaning to some operator combinations. Galton and Augusto also present the core of an alternative, interval-based, semantics of the Snoop operators to remedy these problems. Adaikkalavan and Chakravarthy [21] extended this work into a full interval-based version of Snoop, by formulating the operator semantics for the different event contexts in Snoop. They do not, however, investigate the algebraic properties of the resulting algebra. In particular, it is not clear to what extent the desired properties identified by Galton and Augusto are maintained when event contexts are applied.

The basic semantics of our operators is similar to the one proposed by Galton and Augusto, but some details differ. For example, the Snoop negation operator takes three arguments, denoting the non-occuring event and the events marking the start and end of the non-occurrence interval, respectively (corresponding to $(A;B)-D$ in our algebra). We also include temporal restriction as an explicit operator.

The restriction policy that we introduce to permit the algebra to be implemented with limited resources while retaining the desired algebraic properties, is similar to the *recent event context* of Snoop in that from a set of possible detections it considers only the most recent one. There are significant differences though. Event contexts are applied to individual operators in an expression, in order to provide detailed control of how that operator handles situations where there are several ways to form an instance of the composite event. Thus, their main purpose is increased expressiveness, although the recent context can in fact be implemented with bounded resources. Contrasting this, the aim of our restriction policy is to ensure bounded resource requirements while affecting the ordinary operator semantics as little as possible. The restriction policy is applied once to the expression as a whole, and defines what occurrences may be ignored by the detection mechanism in order to being able to statically bound resource usage. There is also a concrete difference in that the recent context consider the constituent event occurrence with latest end time to be the most recent one, and thus the one used to create a composite instance. Our restriction policy, on the other hand, selects a constituent occurrence that maximises the start time of the resulting composite instance.

Solicitor [22] is an interval-based event specification language similar in style to Snoop. It is also similar to our algebra in that it targets real-time systems in particular. The resource requirements of the Solicitor detection mechanism are bounded in the case when a minimum interarrival time is given for each primitive event, and all subexpressions are labelled with an explicit expiration time. In the notation of our algebra, the expiration time property corresponds to requiring that all operators (except temporal restriction) must be directly enclosed by a temporal restriction (e.g., as in $(A;(B+C)_{\tau})_{\tau'}$).

Sánchez et al. have developed ECL (and the equally expressive sublanguage PAR), a specification language for event patterns [23]. The language resembles regular expressions in style, and any PAR pattern can be detected with bounded resources. A central result is that the opposite is true as well, i.e., that every event pattern that can be detected with finite memory, by any method, can be expressed in PAR [24]. A difference compared with our algebra is that PAR normally only detects the first occurrence of an expression. To allow repeated detection, the language includes a repetition operator which restarts the detection procedure every time an occurrence is detected. The result is different from our algebra in that it does not detect partially overlapping occurrences. E.g., the pattern $A;(B;C)$ and the primitive occurrences A, B, A, C, B and C result in two detections by our algebra but only one (at the first C occurrence) with repeated detection in PAR.

The event stream algebra CESAR, with the associated event processing system Cayuga, explicitly aims at combining simple, well-defined semantics and efficient implementation [25,26]. Starting from a limited event algebra, operators supporting iteration, aggregation and parameterisation are added. The result is a highly expressive algebra with reasonably simple operator semantics. However, no algebraic properties are presented, and although efficient, the implementation can not provide resource bounds.

6 Conclusions

Many event-based systems are concerned with the occurrences of certain event patterns rather than individual event occurrences. Deciding on an appropriate technique to specify such patterns involves finding a suitable tradeoff between expressiveness, efficiency and simplicity of use, for the application in question. For example, resource-constrained applications such as embedded systems typically require a low and predictable overhead in terms of time and memory.

We have presented a novel algebra for event pattern specification, that satisfies a number of laws that intuitively ought to hold for the algebra operators. These laws facilitate formal as well as informal reasoning about the algebra and the behaviour of a system in which it is included, and justify the algebra semantics by showing to what extent it complies with intuition.

The algebra is defined in two steps. The operators are given a straightforward declarative semantics based on time intervals. To deal with resource issues, a restriction policy is applied that defines a subset of occurrences that should be detected. Conceptually, the restriction policy is applied once to the whole expression, which simplifies the overall semantics, but it can also be applied to each subexpression without affecting the overall result. This allows the formulation of an efficient detection algorithm that correctly detects any expression with bounded memory.

The focus so far has been to lay the formal foundations of an event algebra that combines bounded resources and explicit algebraic properties. However, many aspects of event pattern detection are yet to be addressed, including for example operators defining periodical and aperiodic occurrences, filtering of event occurrences based on the values they carry, managing out-of-order arrivals in distributed systems, and many concerns related to the implementation and packaging of the algebra in a form that is easily usable by applications in the intended domain. Future work includes investigating to what extent these and other aspects can be addressed in the context of our algebra without compromising the results achieved so far.

Another line of future work is to use the algebraic laws as the basis for expression transformations, for example to improve the worst case execution time of the detection of a given expression. There are also initial results on scheduling theory for embedded real-time systems where some parts are triggered by complex event patterns defined by this algebra [27] that we want to pursue further.

Acknowledgements

This work was partially funded by the Swedish National Graduate School in Computer Science (CUGS) and the Swedish Foundation for Strategic Research via the strategic research centre Progress.

Appendix A: Proof of Theorem 15

Theorem 15 If $rem(S, S')$ and $rem(T, T')$ hold, than for any event stream U and $\tau \in \mathcal{T}$ the following implications hold:

- $rem(\text{dis}(S',T'),\ U) \quad \Rightarrow \quad rem(\text{dis}(S,T),\ U)$
- $rem(\text{con}(S',T'),\ U) \quad \Rightarrow \quad rem(\text{con}(S,T),\ U)$
- $rem(\text{neg}(S',T'),\ U) \quad \Rightarrow \quad rem(\text{neg}(S,T),\ U)$
- $rem(\text{seq}(S',T'),\ U) \quad \Rightarrow \quad rem(\text{seq}(S,T),\ U)$
- $rem(\text{tim}(S',\tau),\ U) \quad \Rightarrow \quad rem(\text{tim}(S,\tau),\ U)$

PROOF. We prove each implication in a separate case:

Disjunction case: Assume $rem(\text{dis}(S',T'),U)$. Then, for any $u \in U$ we have $u \in \text{dis}(S',T')$ and thus $u \in S' \cup T'$. Since $S' \subseteq S$ and $T' \subseteq T$, we have $u \in S \cup T$, implying $u \in \text{dis}(S,T)$. Thus $U \subseteq \text{dis}(S,T)$, which satisfies the first constraint in the definition of *rem*.

Next, take an arbitrary $u \in \text{dis}(S,T)$. Then $u \in S \cup T$ and according to the definition of *rem* there must exist an $u' \in S' \cup T'$ such that $\text{start}(u) \leq \text{start}(u')$ and $\text{end}(u') = \text{end}(u)$. We have $u' \in \text{dis}(S',T')$ and thus $rem(\text{dis}(S',T'),U)$ implies that there exists an $u'' \in U$ with $\text{start}(u') \leq \text{start}(u'')$ and $\text{end}(u'') =$

end(u'). Since this means that start$(u) \leq$ start(u'') and end$(u'')=$ end(u), the second constraint in the definition of *rem* is satisfied.

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

Conjunction case: Assume *rem*(con$(S', T'), U)$. Then, for any $u \in U$ we have $u \in$ con(S', T') and thus $u = s \oplus t$ 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 $u \in$ con(S, T) and thus $U \subseteq$ con(S, T).

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

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

Negation case: Assume *rem*(neg$(S', T'), U)$. Then, for any $u \in U$ we have $u \in$ 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 start$(u) \leq$ start(t) and end$(t) \leq$ end(u), then there must exist some $t' \in T'$ such that start$(t) \leq$ start(t') and end$(t') =$ end(t) which contradicts the fact that $u \in$ neg(S', T'). Since no such t can exist, we have $u \in$ neg(S, T) and thus $U \subseteq$ neg(S, T).

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

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

Sequence case: Assume *rem*(seq$(S', T'), U)$. Then, for any $u \in U$ we have $u \in$ seq(S', T') and thus $u = s \oplus t$ with $s \in S'$, $t \in T'$ and end$(s) <$ start(t). By the subset requirement in the definition of *rem*, $s \in S$ and $t \in T$, so $u \in$ seq(S, T) and thus $U \subseteq$ seq(S, T).

Next, take an arbitrary $u \in$ seq(S, T). Then $u = s \oplus t$ such that $s \in S$, $t \in T$ and end$(s) <$ start(t). 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 $u' = s' \oplus t'$. 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 $\mathrm{start}(u) \leq \mathrm{start}(u'')$ and $\mathrm{end}(u'') = \mathrm{end}(u)$, which satisfies the second constraint in the definition of *rem*.

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

Temporal restriction case: Assume $\mathit{rem}(\mathrm{tim}(S', \tau), U)$. For any $u \in U$ we have $u \in \mathrm{tim}(S', \tau)$ and thus $u \in S'$ and $\mathrm{end}(u) - \mathrm{start}(u) \leq \tau$. From the subset requirement in the definition of *rem*, we know that $u \in S$, which means that $u \in \mathrm{tim}(S, \tau)$ and thus $U \subseteq \mathrm{tim}(S, \tau)$.

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

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

$\square$

Appendix B: Proof of Theorems 21–23

To simplify the proofs for negation, we introduce the following predicate.

Definition 42 *For an event stream S, and time instants $\tau, \tau' \in \mathcal{T}$, define $\mathit{empty}(S, \tau, \tau')$ to hold if $\neg\exists s(s \in S \wedge \tau \leq \mathrm{start}(s) \wedge \mathrm{end}(s) \leq \tau')$.*

Proposition 43

i. $a \in [\![A - B]\!] \;\Leftrightarrow\; (a \in [\![A]\!] \wedge \mathit{empty}([\![B]\!], \mathrm{start}(a), \mathrm{end}(a)))$.
ii. $\mathit{empty}(S \cup S', \tau, \tau') \;\Leftrightarrow\; (\mathit{empty}(S, \tau, \tau') \wedge \mathit{empty}(S', \tau, \tau'))$
iii. $(\tau_1 \leq \tau_1' \leq \tau_2' \leq \tau_2 \wedge \mathit{empty}(S, \tau_1, \tau_2)) \;\Rightarrow\; \mathit{empty}(S, \tau_1', \tau_2')$

PROOF. The properties follow straightforwardly from the definition and the operator semantics. □

In the proofs below, $\equiv^{23}$ denotes that the equivalence follows from law number 23, etc. Similarly, $=^{i}$ or $\Leftrightarrow^{ii}$ denotes that the equivalence is based on the corresponding property in Proposition 43, and $=^{\oplus}$ is based on the properties of $\oplus$ from Definition 3.

Theorem 21 For event expressions A, B and C, the following laws hold:

1.	$A \vee A \equiv A$	6.	$A;(B;C) \equiv (A;B);C$
2.	$A \vee B \equiv B \vee A$	7.	$(A \vee B)+C \equiv (A+C) \vee (B+C)$
3.	$A+B \equiv B+A$	∗8.	$A+(B \vee C) \equiv (A+B) \vee (A+C)$
4.	$A \vee (B \vee C) \equiv (A \vee B) \vee C$	9.	$(A \vee B);C \equiv (A;C) \vee (B;C)$
5.	$A+(B+C) \equiv (A+B)+C$	10.	$A;(B \vee C) \equiv (A;B) \vee (A;C)$

PROOF.

1. $[\![A \vee A]\!] = \text{dis}([\![A]\!], [\![A]\!]) = [\![A]\!] \cup [\![A]\!] = [\![A]\!]$
2. $[\![A \vee B]\!] = \text{dis}([\![A]\!], [\![B]\!]) = \text{dis}([\![B]\!], [\![A]\!]) = [\![B \vee A]\!]$
3. $[\![A+B]\!] = \text{con}([\![A]\!], [\![B]\!]) =^{\oplus} \text{con}([\![B]\!], [\![A]\!]) = [\![B+A]\!]$
4. $[\![A \vee (B \vee C)]\!] = [\![A]\!] \cup [\![B]\!] \cup [\![C]\!] = [\![(A \vee B) \vee C]\!]$
5. $[\![A+(B+C)]\!] = \text{con}([\![A]\!], \text{con}([\![B]\!], [\![C]\!])) =$
   $\{a \oplus (b \oplus c) \mid a \in [\![A]\!] \wedge b \in [\![B]\!] \wedge c \in [\![C]\!]) =^{\oplus}$
   $\{(a \oplus b) \oplus c \mid a \in [\![A]\!] \wedge b \in [\![B]\!] \wedge c \in [\![C]\!]) = [\![(A+B)+C]\!]$
6. $[\![A;(B;C)]\!] = \{a \oplus e \mid a \in [\![A]\!] \wedge \text{end}(a) < \text{start}(e) \wedge$
   $e \in \{b \oplus c \mid b \in [\![B]\!] \wedge c \in [\![C]\!] \wedge \text{end}(b) < \text{start}(c)\}\} =$
   $\{a \oplus (b \oplus c) \mid a \in [\![A]\!] \wedge b \in [\![B]\!] \wedge c \in [\![C]\!] \wedge \text{end}(a) < \text{start}(b) \wedge$
   $\text{end}(b) < \text{start}(c)\} =^{\oplus}$
   $\{(a \oplus b) \oplus c) \mid a \in [\![A]\!] \wedge b \in [\![B]\!] \wedge c \in [\![C]\!] \wedge \text{end}(a) < \text{start}(b) \wedge$
   $\text{end}(b) < \text{start}(c)\} =$
   $\{e \oplus c \mid e \in \{a \oplus b \mid a \in [\![A]\!] \wedge b \in [\![B]\!] \wedge \text{end}(a) < \text{start}(b)\} \wedge$
   $c \in [\![C]\!] \wedge \text{end}(e) < \text{start}(c)\} = [\![(A;B);C]\!]$
7. $[\![(A \vee B)+C]\!] = \text{con}(\text{dis}([\![A]\!], [\![B]\!]), [\![C]\!]) = \text{con}(([\![A]\!] \cup [\![B]\!]), [\![C]\!]) =$
   $\{e \oplus c \mid e \in [\![A]\!] \cup [\![B]\!] \wedge c \in [\![C]\!]\} =$
   $\{a \oplus c \mid a \in [\![A]\!] \wedge c \in [\![C]\!]\} \cup \{b \oplus c \mid b \in [\![A]\!] \wedge c \in [\![C]\!]\} =$
   $\text{con}([\![A]\!], [\![C]\!]) \cup \text{con}([\![B]\!], [\![C]\!]) = [\![(A+C) \vee (B+C)]\!]$
8. $A+(B \vee C) \equiv^{3} (B \vee C)+A \equiv^{7} (B+A) \vee (C+A) \equiv^{3} (A+B) \vee (A+C)$
9. $[\![(A \vee B);C]\!] = \{e \cup c \mid e \in [\![A]\!] \cup [\![B]\!] \wedge c \in [\![C]\!] \wedge \text{end}(e) < \text{start}(c)\} =$
   $\{a \cup c \mid a \in [\![A]\!] \wedge c \in [\![C]\!] \wedge \text{end}(a) < \text{start}(c)\} \cup$
   $\{b \cup c \mid b \in [\![B]\!] \wedge c \in [\![C]\!] \wedge \text{end}(b) < \text{start}(c)\} = [\![(A;C) \vee (B;C)]\!]$
10. $[\![A;(B \vee C)]\!] = \{a \oplus e \mid a \in [\![A]\!] \wedge e \in [\![B]\!] \cup [\![C]\!] \wedge \text{end}(a) < \text{start}(e)\} =$
   $\{a \oplus b \mid a \in [\![A]\!] \wedge b \in [\![B]\!] \wedge \text{end}(a) < \text{start}(b)\} \cup$
   $\{a \oplus c \mid a \in [\![A]\!] \wedge c \in [\![C]\!] \wedge \text{end}(a) < \text{start}(c)\} = [\![(A;B) \vee (A;C)]\!]$

$\square$

Theorem 22 For event expressions A, B and C, the following laws hold:

11. $(A\vee B)-C \equiv (A-C)\vee(B-C)$	$*$15. $(A-B)-B \equiv A-B$
12. $(A+B)-C \equiv ((A-C)+B)-C$	$*$16. $(A-B)-C \equiv (A-C)-B$
$*$13. $(A+B)-C \equiv (A+(B-C))-C$	17. $(A;B)-C \equiv ((A-C);B)-C$
14. $(A-B)-C \equiv A-(B\vee C)$	18. $(A;B)-C \equiv (A;(B-C))-C$

PROOF.

11. $[\![(A\vee B)-C]\!] =^{i} \{e \mid e \in [\![A]\!] \cup [\![B]\!] \wedge \mathit{empty}([\![C]\!], \mathrm{start}(e), \mathrm{end}(e))\} =$
$\{a \mid a\in[\![A]\!] \wedge \mathit{empty}([\![C]\!], \mathrm{start}(a), \mathrm{end}(a))\} \cup$
$\{b \mid b\in[\![B]\!] \wedge \mathit{empty}([\![C]\!], \mathrm{start}(b), \mathrm{end}(b))\} =^{i}$
$[\![(A-C)]\!] \cup [\![(B-C)]\!] = [\![(A-C)\vee(B-C)]\!]$
12. $e\in[\![((A-C)+B)-C]\!] \Leftrightarrow^{i} e\in[\![(A-C)+B]\!] \wedge \mathit{empty}([\![C]\!], \mathrm{start}(e), \mathrm{end}(e)) \Leftrightarrow$
$e=a \oplus b \wedge a\in[\![A-C]\!] \wedge b\in[\![B]\!] \wedge \mathit{empty}([\![C]\!], \mathrm{start}(e), \mathrm{end}(e)) \Leftrightarrow^{i}$
$e=a \oplus b \wedge a\in[\![A]\!] \wedge b\in[\![B]\!] \wedge \mathit{empty}([\![C]\!], \mathrm{start}(e), \mathrm{end}(e)) \wedge$
$\mathit{empty}([\![C]\!], \mathrm{start}(a), \mathrm{end}(a)) \Leftrightarrow^{iii}$
$e=a \oplus b \wedge a\in[\![A]\!] \wedge b\in[\![B]\!] \wedge \mathit{empty}([\![C]\!], \mathrm{start}(e), \mathrm{end}(e)) \Leftrightarrow$
$e\in[\![A+B]\!] \wedge \mathit{empty}([\![C]\!], \mathrm{start}(e), \mathrm{end}(e)) \Leftrightarrow^{i} e\in[\![(A+B)-C]\!]$
13. $(A+B)-C \equiv^{3} (B+A)-C \equiv^{12} ((B-C)+A)-C \equiv^{3} (A+(B-C))-C$
14. $a\in[\![(A-B)-C]\!] \Leftrightarrow^{i} a\in[\![A-B]\!] \wedge \mathit{empty}([\![C]\!], \mathrm{start}(a), \mathrm{end}(a)) \Leftrightarrow^{i}$
$a\in[\![A]\!] \wedge \mathit{empty}([\![B]\!], \mathrm{start}(a), \mathrm{end}(a)) \wedge \mathit{empty}([\![C]\!], \mathrm{start}(a), \mathrm{end}(a)) \Leftrightarrow^{ii}$
$a\in[\![A]\!] \wedge \mathit{empty}([\![B]\!] \cup [\![C]\!], \mathrm{start}(a), \mathrm{end}(a)) \Leftrightarrow^{i} a\in[\![A-(B\vee C)]\!]$
15. $(A-B)-B \equiv^{14} A-(B\vee B) \equiv^{1} A-B$
16. $(A-B)-C \equiv^{14} A-(B\vee C) \equiv^{2} A-(C\vee B) \equiv^{14} (A-C)-B$
17. $e\in[\![((A-C);B)-C]\!] \Leftrightarrow^{i} e\in[\![(A-C);B]\!] \wedge \mathit{empty}([\![C]\!], \mathrm{start}(e), \mathrm{end}(e)) \Leftrightarrow$
$e=a \oplus b \wedge \mathrm{end}(a)<\mathrm{start}(b) \wedge a\in[\![A-C]\!] \wedge b\in[\![B]\!] \wedge$
$\mathit{empty}([\![C]\!], \mathrm{start}(a), \mathrm{end}(b)) \Leftrightarrow^{i}$
$e=a \oplus b \wedge \mathrm{end}(a)<\mathrm{start}(b) \wedge a\in[\![A]\!] \wedge b\in[\![B]\!] \wedge$
$\mathit{empty}([\![C]\!], \mathrm{start}(a), \mathrm{end}(b)) \wedge \mathit{empty}([\![C]\!], \mathrm{start}(a), \mathrm{end}(a)) \Leftrightarrow^{iii}$
$e=a \oplus b \wedge \mathrm{end}(a)<\mathrm{start}(b) \wedge a\in[\![A]\!] \wedge b\in[\![B]\!] \wedge$
$\mathit{empty}([\![C]\!], \mathrm{start}(a), \mathrm{end}(b)) \Leftrightarrow$
$e\in[\![A;B]\!] \wedge \mathit{empty}([\![C]\!], \mathrm{start}(e), \mathrm{end}(e)) \Leftrightarrow^{i} e\in[\![(A;B)-C]\!]$
18. $e\in[\![(A;(B-C))-C]\!] \Leftrightarrow^{i} e\in[\![A;(B-C)]\!] \wedge \mathit{empty}([\![C]\!], \mathrm{start}(e), \mathrm{end}(e)) \Leftrightarrow$
$e=a \oplus b \wedge \mathrm{end}(a)<\mathrm{start}(b) \wedge a\in[\![A]\!] \wedge b\in[\![B-C]\!] \wedge$
$\mathit{empty}([\![C]\!], \mathrm{start}(a), \mathrm{end}(b)) \Leftrightarrow^{i}$
$e=a \oplus b \wedge \mathrm{end}(a)<\mathrm{start}(b) \wedge a\in[\![A]\!] \wedge b\in[\![B]\!] \wedge$
$\mathit{empty}([\![C]\!], \mathrm{start}(a), \mathrm{end}(b)) \wedge \mathit{empty}([\![C]\!], \mathrm{start}(b), \mathrm{end}(b)) \Leftrightarrow^{iii}$
$e=a \oplus b \wedge \mathrm{end}(a)<\mathrm{start}(b) \wedge a\in[\![A]\!] \wedge b\in[\![B]\!] \wedge$
$\mathit{empty}([\![C]\!], \mathrm{start}(a), \mathrm{end}(b)) \Leftrightarrow$
$e\in[\![A;B]\!] \wedge \mathit{empty}([\![C]\!], \mathrm{start}(e), \mathrm{end}(e)) \Leftrightarrow^{i} e\in[\![(A;B)-C]\!]$

□

Theorem 23 For event expressions A and B, and $\tau \in \mathcal{T}$, the following laws hold:

19.	$(A \vee B)_\tau \equiv A_\tau \vee B_\tau$	24.	$(A;B)_\tau \equiv (A_\tau;B)_\tau$
20.	$(A+B)_\tau \equiv (A_\tau + B)_\tau$	25.	$(A;B)_\tau \equiv (A;B_\tau)_\tau$
∗21.	$(A+B)_\tau \equiv (A+B_\tau)_\tau$	26.	$A \equiv A_\tau$ if $A \in \mathcal{P}$
22.	$(A-B)_\tau \equiv A_\tau - B$	27.	$(A_\tau)_{\tau'} \equiv A_{\min(\tau,\tau')}$
23.	$(A-B)_\tau \equiv (A-B_\tau)_\tau$	∗28.	$(A_\tau)_{\tau'} \equiv (A_{\tau'})_\tau$

PROOF.

19. $[\![(A \vee B)_\tau]\!] = \{e \mid e \in A \cup B \wedge \text{end}(e) - \text{start}(e) \leq \tau\} =$
$\{a \mid a \in A \wedge \text{end}(a) - \text{start}(a) \leq \tau\} \cup \{b \mid b \in B \wedge \text{end}(b) - \text{start}(b) \leq \tau\} =$
$[\![A_\tau]\!] \cup [\![B_\tau]\!] = [\![A_\tau \vee B_\tau]\!]$
20. $e \in [\![(A_\tau + B)_\tau]\!] \Leftrightarrow e \in [\![A_\tau + B]\!] \wedge \text{end}(e) - \text{start}(e) \leq \tau \Leftrightarrow$
$e = a \oplus b \wedge a \in [\![A_\tau]\!] \wedge b \in [\![B]\!] \wedge \text{end}(e) - \text{start}(e) \leq \tau \Leftrightarrow$
$e = a \oplus b \wedge a \in [\![A]\!] \wedge \text{end}(a) - \text{start}(a) \leq \tau \wedge b \in [\![B]\!] \wedge \text{end}(e) - \text{start}(e) \leq \tau$.
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:
$e = a \oplus b \wedge a \in [\![A]\!] \wedge b \in [\![B]\!] \wedge \text{end}(e) - \text{start}(e) \leq \tau \Leftrightarrow$
$e \in [\![A_\tau + B]\!] \wedge \text{end}(e) - \text{start}(e) \leq \tau \Leftrightarrow e \in [\![(A+B)_\tau]\!]$.
21. $(A+B)_\tau \equiv^3 (B+A)_\tau \equiv^{20} (B_\tau + A)_\tau \equiv^3 (A + B_\tau)_\tau$
22. $[\![(A-B)_\tau]\!] = \{a \mid a \in [\![A-B]\!] \wedge \text{end}(a) - \text{start}(a) \leq \tau\} =$
$\{a \mid a \in [\![A]\!] \wedge \textit{empty}([\![B]\!], \text{start}(a), \text{end}(a)) \wedge \text{end}(a) - \text{start}(a) \leq \tau\} =$
$\{a \mid a \in [\![A_\tau]\!] \wedge \textit{empty}([\![B]\!], \text{start}(a), \text{end}(a))\} = [\![A_\tau - B]\!]$
23. $[\![(A-B_\tau)_\tau]\!] = \{a \mid a \in [\![A]\!] \wedge \text{end}(a) - \text{start}(a) \leq \tau \wedge$
$\neg\exists b(b \in [\![B_\tau]\!] \wedge \text{start}(a) \leq \text{start}(b) \wedge \text{end}(b) \leq \text{end}(a))\} =$
$\{a \mid a \in [\![A]\!] \wedge \text{end}(a) - \text{start}(a) \leq \tau \wedge \neg\exists b(b \in [\![B]\!] \wedge$
$\text{start}(a) \leq \text{start}(b) \wedge \text{end}(b) \leq \text{end}(a) \wedge \text{end}(b) - \text{start}(b) \leq \tau)\}$
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 affecting the set. Thus, the set above is equivalent to:
$\{a \mid a \in [\![A]\!] \wedge \text{end}(a) - \text{start}(a) \leq \tau \wedge$
$\neg\exists b(b \in [\![B]\!] \wedge \text{start}(a) \leq \text{start}(b) \wedge \text{end}(b) \leq \text{end}(a))\} = [\![(A-B)_\tau]\!]$.
24. $[\![(A;B_\tau)_\tau]\!] =$
$\{a \oplus b \mid a \in [\![A]\!] \wedge b \in [\![B_\tau]\!] \wedge \text{end}(a) < \text{start}(b) \wedge \text{end}(b) - \text{start}(a) \leq \tau\} =$
$\{a \oplus b \mid a \in [\![A]\!] \wedge b \in [\![B]\!] \wedge \text{end}(b) - \text{start}(b) \leq \tau \wedge \text{end}(a) < \text{start}(b) \wedge$
$\text{end}(b) - \text{start}(a) \leq \tau\}$
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 constraint can be dropped without changing the set. Thus, the set

above is equivalent to $\{a \oplus b \mid a \in [\![A]\!] \wedge b \in [\![B]\!] \wedge \text{end}(a) < \text{start}(b) \wedge \text{end}(b) - \text{start}(a) \leq \tau\} = [\![(A;B)_\tau]\!]$

25. $[\![(A_\tau;B)_\tau]\!] =$
$\{a \oplus b \mid a \in [\![A_\tau]\!] \wedge b \in [\![B]\!] \wedge \text{end}(a) < \text{start}(b) \wedge \text{end}(b) - \text{start}(a) \leq \tau\} =$
$\{a \oplus b \mid a \in [\![A]\!] \wedge \text{end}(a) - \text{start}(a) \leq \tau \wedge b \in [\![B]\!] \wedge \text{end}(a) < \text{start}(b) \wedge \text{end}(b) - \text{start}(a) \leq \tau\}$
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 constraint can be dropped without changing the set. Thus, the set above is equivalent to $\{a \oplus b \mid a \in [\![A]\!] \wedge b \in [\![B]\!] \wedge \text{end}(a) < \text{start}(b) \wedge \text{end}(b) - \text{start}(a) \leq \tau\} = [\![(A;B)_\tau]\!]$
26. $A \in \mathcal{P}$ implies that $\text{end}(a) - \text{start}(a) = 0$ for any $a \in [\![A]\!]$, which means that $[\![A]\!] = [\![A_\tau]\!]$.
27. $[\![(A_\tau)_{\tau'}]\!] = \{a \mid a \in [\![A]\!] \wedge \text{end}(a) - \text{start}(a) \leq \tau \wedge \text{end}(a) - \text{start}(a) \leq \tau'\} = \{a \mid a \in [\![A]\!] \wedge \text{end}(a) - \text{start}(a) \leq \min(\tau, \tau')\} = [\![A_{\min(\tau,\tau')}]\!]$
28. $(A_\tau)_{\tau'} \equiv^{23} A_{\min(\tau,\tau')} \equiv A_{\min(\tau,\tau')} \equiv^{23} (A_{\tau'})_\tau$

□

Appendix C: Proof of Lemma 33

Lemma 33 Assume that *state*(i, τ) held at the start of the current tick and that *pcorr*(n, τ) and *acorr*(n, τ) hold for all $1 \leq n < i$. Then *state*$(i, \tau + 1)$, *pcorr*(i, τ) and *acorr*(i, τ) hold after executing the loop body once.

PROOF. The proof is organised in four parts. First, we consider *state*, then the two criteria that are required for *pcorr* to hold (see Definition 31), and finally *acorr* is addressed.

For *state*, we see that *state*$(i, \tau + 1)$ holds trivially if E^i is primitive, a disjunction or a temporal restriction (Definition 32), and thus we consider the remaining operators:

Case $E^i = E^j + E^k$: In the case $a_j = \langle\rangle$ the l_i variable remains unchanged, which is consistent with *state*$(i, \tau{+}1)$. If $a_j \neq \langle\rangle$ then $\text{end}(a_j) = \tau$ according to the assumption *acorr*(j, τ). Then, the first conditional in the conjunction part ensures that l_i contains an instance consistent with *state*$(i, \tau{+}1)$. Similarly, the second conditional ensures the correctness of r_i.

Case $E^i = E^j - E^k$: The first conditional in the negation part ensures that t_i contains the value specified by *state*$(i, \tau{+}1)$.

Case $E^i = E^j;E^k$: The l_i variable is updated by the last conditional in the sequence part, and the proof is identical to that in the conjunction case above.

For the second criterion in the definition of *state*, let t be an arbitrary element in S_k such that $\{e \mid e \in \mathcal{A}(j) \wedge \text{end}(e) < t\}$ is non-empty. We consider two cases: If t was in S_k at the start of the current tick, then $\mathit{state}(i, \tau)$ ensures that Q_i contained an element from $\{e \mid e \in \mathcal{A}(j) \wedge \text{end}(e) < t\}$ with maximum start time at the start of this tick. If t was not in S_k at the start of the current tick, then $\mathit{pcorr}(k, \tau)$ implies that $t = \tau$, and then $\mathit{state}(i, \tau)$ ensures that l_i contained an element from $\{e \mid e \in \mathcal{A}(j) \wedge \text{end}(e) < t\}$ with maximum start time at the start of this tick. Thus, in both cases, $Q_i \cup l_i$ contained such an element at the start of this tick. We can see that the inner foreach construct in the sequence part assigns an element from this set to e', and thus it is added to Q' and finally to Q_i.

For *pcorr*, let S denote the content of S_i at the start of the current time tick. We focus first on the first criterion in the definition of *pcorr*, which requires that we have $a_i = \langle\rangle$, $\text{start}(a_i) = \tau$ or $\text{start}(a_i) \in S$. For $E^i \in \mathcal{P}$, we know that a_i is a primitive event instance, and thus $\text{start}(a_i) = \text{end}(a_i) = \tau$. For the operators, we note that the first criterion of $\mathit{pcorr}(i, \tau)$ holds trivially when $a_i = \langle\rangle$ or $\text{start}(a_i) = \tau$, so we consider only the case when $a_i \neq \langle\rangle$ and $\text{start}(a_i) \neq \tau$.

Case $E^i = E^j \vee E^k$: If $a_i = a_j$, we know according to $\mathit{pcorr}(j, \tau)$ that $\text{start}(a_i)$ was in S_j at the start of this tick. Since $S_j \subseteq S_i$ must hold at the start of each tick (at initialisation, and after each subsequent assignment of S_i), this implies $\text{start}(a_i) \in S$. If $a_i = a_k$, the same result is implied by $\mathit{pcorr}(k, \tau)$ and $S_k \subseteq S_i$.

Case $E^i = E^j + E^k$: From the two assignments of a_i in the conjunction part where $a_i \neq \langle\rangle$, we can see that the start time of a_i must be equal to the start time of a_j, r_i, l_i or a_k. For a_j and a_k we can reuse the disjunction proof above. If $\text{start}(a_i) = \text{start}(l_i)$, we have to consider two subcases: If l_i was updated in this tick, we have $l_i = a_j$ and we can reuse the proof above. If l_i remained unchanged, then $l_i \neq \langle\rangle$ ensures that the current tick is not the first, and the assignment of S_i in the previous step implies that $\text{start}(l_i) \in S$. The proof for the final case $\text{start}(a_i) = \text{start}(r_i)$ is analogous.

Case $E^i = E^j - E^k$: Analogous to the $a_i = a_j$ case in the disjunction proof.

Case $E^i = E^j ; E^k$: Since $a_i \neq \langle\rangle$, we have $a_i = a_k \oplus e'$ where $\text{start}(a_i) = \text{start}(e')$ and e' was in Q_i or l_i at the start of this tick. This implies that the current tick is not the first, and the assignment of S_i in the previous step ensures that $\text{start}(e') \in S$.

Case $E^i = E^j_{\tau'}$: Analogous to the $a_i = a_j$ case in the disjunction proof.

Next, we consider the second criterion in the definition of *pcorr*, namely that $\forall t \; (t \in S_i \Rightarrow (t = \tau \vee t \in S))$. As previously, S denotes the content of S_i at the start of the current time tick. The property trivially holds for $E^i \in \mathcal{P}$, since this implies $S = \emptyset$. For the operators, consider an arbitrary $t \in S_i$ such that $t \neq \tau$.

Case $E^i = E^j \vee E^k$: Since $S_i = S_j \cup S_k$, we must have $t \in S_j$ or $t \in S_k$. If $t \in S_j$, then $\mathit{pcorr}(j, \tau)$ implies that t was in S_j at the start of this tick. Since $S_j \subseteq S_i$ holds at the start of each tick, this implies $t \in S$. If $t \in S_k$, the same result follows from $\mathit{pcorr}(k, \tau)$ and $S_k \subseteq S_i$.

Case $E^i = E^j{+}E^k$: The assignment of S_i in the conjunction part implies that $t \in S_j \cup S_k \cup \{\text{start}(l_i), \text{start}(r_i)\}$. If $t \in S_j \cup S_k$, we can reuse the disjunction proof above. If $t = \text{start}(l_i)$ we consider two subcases: If l_i remained unchanged in this tick, then the assignment of S_i in the previous tick ensures than $t \in S$. If l_i was updated, we have $l_i = a_j$, and then $\mathit{pcorr}(j, \tau)$ ensures that t was in S_j at the start of this tick. As shown above, this implies $t \in S$. The proof for the final case $t = \text{start}(r_i)$ is analogous.

Case $E^i = E^j{-}E^k$: Analogous to the $t \in S_j$ case in the disjunction proof.

Case $E^i = E^j;E^k$: The assignment of S_i in the sequence part implies that $t \in S_j$, $t = \text{start}(l_i)$ or $t \in \{\text{start}(e) \mid e \in Q_i\}$. For the two first cases we can reuse the proof for conjunction. If $t = \text{start}(e)$ where $e \in Q_i$, then e was added to Q' in the nested foreach constructs, which means that e was in $Q_i \cup l_i$ at the start of this tick (so this is not the first tick). Then, the assignment of S_i in the previous tick ensures than $t \in S$.

Case $E^i = E^j_{\tau'}$: Analogous to the $t \in S_j$ case in the disjunction proof.

Finally, for *acorr*, we consider the following six cases:

Case $E^i \in \mathcal{P}$: If $a_i = \langle\rangle$, then there is no $e \in [\![E^i]\!]$ with $\text{end}(e) = \tau$, and thus $\mathit{valid}(a_i, [\![E^i]\!], \tau)$ holds. If $a_i \neq \langle\rangle$, we have $a_i \in [\![E^i]\!]$ and $\text{end}(a_i) = \tau$, and since the elements of $[\![E^i]\!]$ have distinct end times according to Definition 6, $\mathit{valid}(a_i, [\![E^i]\!], \tau)$ holds.

Case $E^i = E^j \vee E^k$: The detection algorithm ensures that $\text{start}(a_j) \leq \text{start}(a_i)$ and $\text{start}(a_k) \leq \text{start}(a_i)$. If $a_i = \langle\rangle$, we have $\text{start}(a_i) = -1$ which implies that $a_k = a_j = \langle\rangle$ so there is no $e \in \text{dis}(\mathcal{A}(j), \mathcal{A}(k))$ with $\text{end}(e) = \tau$. If $a_i \neq \langle\rangle$, we clearly have $a_i \in \text{dis}(\mathcal{A}(j), \mathcal{A}(k))$ and there can be no element in this set with end time τ and start time later than $\text{start}(a_i)$.

Case $E^i = E^j{+}E^k$: After executing the first two conditionals in the conjunction part $\text{start}(a_j) \leq \text{start}(l_i)$ and $\text{start}(a_k) \leq \text{start}(r_i)$ hold. If $a_i = \langle\rangle$, then the guard of the third conditional was satisfied, and there can be no instance in $\text{con}(\mathcal{A}(j), \mathcal{A}(k))$ with end time τ, which concludes the proof. If $a_i \neq \langle\rangle$, then the guard of the third conditional failed, and the inner conditional ensures that $a_i \in \mathrm{con}(\mathcal{A}(j), \mathcal{A}(k))$. For an arbitrary $e \in \mathrm{con}(\mathcal{A}(j), \mathcal{A}(k))$ with $\mathrm{end}(e) = \tau$, we must have $e = e' \oplus a_k$ or $e = a_j \oplus e'$ where $e' \in \mathcal{A}(j) \cup \mathcal{A}(k)$ and $\mathrm{end}(e') \leq \tau$. However, the inner conditional ensures that $\mathrm{start}(a_j) \leq \mathrm{start}(a_i)$ and $\mathrm{start}(a_k) \leq \mathrm{start}(a_i)$ which implies $\mathrm{start}(e) \leq \mathrm{start}(a_i)$, and thus there is no $e \in \mathrm{con}(\mathcal{A}(j), \mathcal{A}(k))$ with $\mathrm{end}(e) = \tau$ and $\mathrm{start}(a_i) < \mathrm{start}(e)$.

Case $E^i = E^j – E^k$: Reusing the proof for *state* above, we know that $\mathit{state}(i, \tau{+}1)$ holds after the first conditional in the negation part. If $a_i = \langle\rangle$, then the guard of the second conditional failed, implying that either $a_j = \langle\rangle$ or there exists an e in $\mathcal{A}(k)$ with $\mathrm{start}(a_j) \leq \mathrm{start}(e)$ and $\mathrm{end}(e) \leq \mathrm{end}(a_j)$. In either case, there is no element in $\mathrm{neg}(\mathcal{A}(j), \mathcal{A}(k))$ with end time τ. If $a_i \neq \langle\rangle$, then $a_i = a_j$ so we have $a_i \in \mathcal{A}(j)$. Furthermore, the guard of the second conditional holds and then according to $\mathit{state}(i, \tau{+}1)$ there is no e in $\mathcal{A}(k)$ with $\mathrm{start}(a_i) \leq \mathrm{start}(e)$ and $\mathrm{end}(e) \leq \mathrm{end}(a_i)$, and thus $a_i \in \mathrm{neg}(\mathcal{A}(j), \mathcal{A}(k))$. Since a_j is the only instance in $\mathcal{A}(j)$ with end time τ, we have $\mathit{valid}(a_i, \mathrm{neg}(\mathcal{A}(j), \mathcal{A}(k), \tau)$.

Case $E^i = E^j ; E^k$: If $a_k = \langle\rangle$ then $e' = \langle\rangle$ after the first foreach construct, and thus $a_i = \langle\rangle$. It also means that there can be no $e \in \mathrm{seq}(\mathcal{A}(j), \mathcal{A}(k))$ with $\mathrm{end}(e) = \tau$, which concludes the proof. If $a_k \neq \langle\rangle$ then $\mathit{pcorr}(k, \tau)$ implies that either $\mathrm{start}(a_k) = \tau$ or $\mathrm{start}(a_k)$ was in S_k at the start of this tick. According to $\mathit{state}(i, \tau)$, this implies that (at the start of this tick) $Q_i \cup \{l_i\}$ contained an element in $\{e \mid e \in \mathcal{A}(j) \wedge \mathrm{end}(e) < \mathrm{start}(a_k)\}$ with maximum start time if that set is non-empty. We consider two subcases: If $e' = \langle\rangle$ after the first foreach contruct, the set was empty, meaning that there can be no element in $e \in \mathrm{seq}(\mathcal{A}(j), \mathcal{A}(k))$ with $\mathrm{end}(e) = \tau$. If $e' \neq \langle\rangle$ after the first foreach contruct, then $a_i = a_k \oplus e'$ ensures that $a_i \in \mathrm{seq}(\mathcal{A}(j), \mathcal{A}(k))$. Furthermore, we know that there is no $e \in \mathcal{A}(j)$ with $\mathrm{end}(e) < \mathrm{start}(a_k)$ and $\mathrm{start}(e') < \mathrm{start}(e)$. Thus, there can be no $e \in \mathrm{seq}(\mathcal{A}(j), \mathcal{A}(k))$ with $\mathrm{start}(a_i) < \mathrm{start}(e)$ and $\mathrm{end}(e) = \tau$.

Case $E^i = E^j_{\tau'}$: If 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, we have $\mathit{valid}(a_j, \mathrm{tim}(\mathcal{A}(j), \tau'), \tau)$. If the conditional fails, then there is no e in $\mathrm{tim}(\mathcal{A}(j), \tau')$ with $\mathrm{end}(e) = \tau$. $\square$

References

[1] H. Kopetz, Event-triggered versus time-triggered real-time systems, Research Report 8/1991, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria (1991).

[2] P. T. Eugster, P. A. Felber, R. Guerraoui, A.-M. Kermarrec, The many faces of publish/subscribe, ACM Comput. Surv. 35 (2) (2003) 114–131.

[3] P. A. Bernstein, Middleware: A model for distributed system services, Communications of the ACM 39 (2) (1996) 86–98.

[4] P. R. Pietzuch, Hermes: A scalable event-based middleware, Ph.D. thesis, University of Cambridge (February 2004).

[5] R. Gruber, B. Krishnamurthy, E. Panagos, The architecture of the READY event notification service, in: P. Dasgupta (Ed.), Proceedings of the 19th IEEE International Conference on Distributed Computing Systems, Middleware Workshop, Austin, TX, USA, 1999.

[6] A. Mok, G. Liu, Efficient run-time monitoring of timing constraints, in: Proceedings of the Third IEEE Real-Time Technology and Applications Symposium (RTAS '97), IEEE, Washington - Brussels - Tokyo, 1997, pp. 252–262.

[7] C. Dousson, Alarm driven supervision for telecommunication networks: II- On-line chronicle recognition, Annals of Telecommunications (1996) 501–508CNET, France Telecom.

[8] C. Liebig, B. Boesling, A. Buchmann, A notification service for next-generation it systems in air traffic control, in: GI-Workshop: Multicast-Protokolle und Anwendungen, Braunschweig, Germany, 1999.

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

[10] S. Chakravarthy, V. Krishnaprasad, E. Anwar, S.-K. Kim, Composite events for active databases: Semantics, contexts and detection, in: 20th International Conference on Very Large Data Bases, Morgan Kaufmann Publishers, Santiago, Chile, 1994, pp. 606–617.

[11] N. Gehani, H. V. Jagadish, O. Shmueli, COMPOSE: A system for composite specification and detection, in: Advanced Database Systems, Vol. 759 of Lecture Notes in Computer Science, Springer, 1993.

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

[13] S. Gatziu, K. R. Dittrich, Detecting composite events in active database systems using petri nets, in: Research Issues in Data Engineering (RIDE '94), IEEE Computer Society Press, Los Alamitos, Ca., USA, 1994, pp. 2–9.

[14] A. Hinze, A. Voisard, A parameterized algebra for event notification services, in: Proceedings of the 9th International Symposium on Temporal Representation and Reasoning (TIME 2002), Springer-Verlag, Manchester, UK, 2002.

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

[16] R. A. Kowalski, M. J. Sergot, A logic-based calculus of events, New Generation Computing 4 (1986) 67–95.

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

[18] J. Carlson, Event pattern detection for embedded systems, Ph.D. thesis, Mälardalen University (June 2007).

[19] J. Carlson, Event Pattern Detection for Embedded Systems — A Resource-efficient Event Algebra, VDM Verlag, 2009.

[20] G. Liu, A. Mok, 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), IEEE, Washington - Brussels - Tokyo, 1998, pp. 199–209.

[21] R. Adaikkalavan, S. Chakravarthy, SnoopIB: Interval-based event specification and detection for active databases, Data Knowledge Engineering 59 (1) (2006) 139–165.

[22] J. Mellin, Resource-predictable and efficient monitoring of events, Ph.D. thesis, Department of Computer Science, University of Skövde (June 2004).

[23] C. Sánchez, H. B. Sipma, M. Slanina, Z. Manna, Final semantics for Event-Pattern Reactive Programs, in: First International Conference in Algebra and Coalgebra in Computer Science (CALCO'05), Vol. 3629 of LNCS, Springer-Verlag, 2005, pp. 364–378.

[24] C. Sánchez, M. Slanina, H. B. Sipma, Z. Manna, Expressive completeness of an event-pattern reactive programming language., in: F. Wang (Ed.), FORTE, Vol. 3731 of Lecture Notes in Computer Science, Springer, 2005, pp. 529–532.

[25] A. Demers, J. Gehrke, M. Hong, M. Riedewald, W. White, A general algebra and implementation for monitoring event streams, Tech. Rep. TR2005-1997, Cornell University (July 2005).

[26] A. Demers, J. Gehrke, M. Hong, M. Riedewald, W. White, Towards expressive publish/subscribe systems, in: Advances in Database Technology - EDBT 2006, 10th International Conference on Extending Database Technology, Vol. 3896 of Lecture Notes in Computer Science, Springer, 2006, pp. 627–644.

[27] J. Carlson, J. Mäki-Turja, M. Nolin, Event-pattern triggered real-time tasks, in: 16th International Conference on Real-Time and Network Systems (RTNS), 2008, pp. 77–85.