An Event Detection Algebra for Reactive Systems

Jan Carlson
jan.carlson@mdh.se

Björn Lisper
bjorn.lisper@mdh.se

Department of Computer Science and Engineering
Mälardalen University, Sweden

ABSTRACT

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

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

Categories and Subject Descriptors

I.1.3 [**Symbolic and Algebraic Manipulation**]: Languages and Systems—*special-purpose algebraic systems*; C.3 [**Computer Systems Organization**]: Special-purpose and Application-based Systems—*real-time and embedded systems*

General Terms

Theory, Performance, Algorithms

Keywords

Event detection, event algebra, resource-efficiency, reactive systems

1. INTRODUCTION

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

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

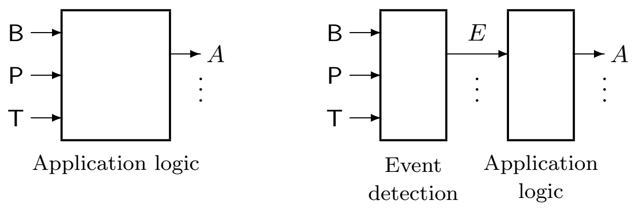

Figure 1: Implicit and explicit detection of composite events

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

The proposed algebra consists of five operators: The *disjunction* of A and B represents that either of A and B occurs, here denoted $A \vee B$. *Conjunction* means that both events have occurred, possibly not simultaneously, and is denoted $A+B$. The *negation*, denoted $A-B$, occurs when there is an occurrence of A during which there is no occurrence of B. 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. The composite event E from the previous example corresponds to the expression $(B;B)_2-(P \vee T)$ in the algebra.

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

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

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

As far as we know, previous research on event algebras has not addressed conformance to algebraic laws. In particular, event algebras suited for systems with limited resources typically exhibit unintuitive semantics and poor algebraic properties. Throughout the paper, proofs have been reduced or left out due to space limitations, and the reader is referred to [3] for full proofs.

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

2. RELATED WORK

The operators of our algebra, as well as the use of interval semantics and restricted detection, are influenced by work in the area of active databases. Snoop [6], Ode [9] and SAMOS [8] are examples of active database systems where an event algebra is used to specify the reactive behaviour. These systems differ primarily in the choice of detection mechanism. SAMOS is based on Petri nets, while Snoop uses event graphs. In Ode, event definitions are equivalent to regular expressions and can be detected by state automata. In the area of active databases, event algebras are often not given a formal semantics, and algebraic properties of the operators are not investigated. Also, resource efficiency is typically not a main concern.

Common to these systems is that they consider composite events to be instantaneous, i.e., an occurrence is associated with a single time instant, normally the time at which it can be detected. Galton and Augusto have shown that this results in unintended semantics for some operation compositions [7]. For example, an occurrence of A followed by B and then C, is accepted as an occurrence of the composite event $B;(A;C)$, since B occurs before the occurrence of $A;C$. They also present the core of an alternative, interval-based, semantics to handle these problems. We use a similar semantic base for our algebra, but we extend it with a restriction policy to allow the algebra to be implemented with limited resources while retaining the desired algebraic properties.

Solicitor [12] is an interval-based event specification language based on Snoop. The language targets real-time systems in particular, and and achieves predictable resource requirements for composite events with an explicit expiration time.

Liu et al. use Real Time Logic to define a system where composite events are expressed as timing constraints and handled by general timing constraint monitoring techniques. They present a mechanism for early detection of timing constraint violation, and show that upper bounds on memory and time can be derived [11].

In middleware platforms, event detection techniques are used to handle high volumes of event occurrences by allowing consumers to subscribe to certain event patterns rather than to single event types. Sánchez et al. present an event correlation language where event expressions are translated into nested Petri net like automata. [14].

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

Knowledge representation techniques use similar operators to reason about event occurrences. Rather than detecting complex events as they occur, they focus on how to express formally the fact that some event has occurred, and on defining inferences rules for this type of statements. Examples include Interval Calculus [1] and Event Calculus [10].

We propose an algebra for which a large class of composite events can be detected with limited resources. The algebra is defined by a simple declarative semantics and we present a number of algebraic laws that facilitate formal reasoning, and supports the claim that the intuitive meaning of the operators is valid also for complex nested expressions. A preliminary version of the algebra, with less useful algebraic properties and with no memory bound, was described in a previous paper [4].

3. DECLARATIVE SEMANTICS

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

Definition 1. The temporal domain $\mathcal{T}$ is the set of all natural numbers.

3.1 Primitive events

We assume that the system has a pre-defined set of primitive event types to which it should be able to react. These events can be external (sampled from the environment or originating from another system) or internal (such as the violation of a condition over the system state, or a timeout), but the detection mechanism does not distinguish between these categories.

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

Definition 2. Let $\mathcal{P}$ be a finite set of identifiers that represent the primitive event types that are of interest to the system. For each identifier $p \in \mathcal{P}$, let $\mathrm{dom}(p)$ denote the domain from which the values of p are taken.

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

Definition 3. If $p \in \mathcal{P}$, $v \in \mathrm{dom}(p)$ and $\tau \in \mathcal{T}$, then the singleton set $\{\langle p, v, \tau\rangle\}$ is a primitive event instance.

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

Definition 4. A primitive event stream is a set of primitive event instances all of which have the same identifier, and different times.

The set of identifiers and the value domains capture static aspects of the system. Instances and event streams, however, are dynamic concepts that describe what happens during a particular scenario. An interpretation is a formal representation of a single scenario, as it describes one of the possible ways in which the primitive events can occur.

Definition 5. An interpretation is a function that maps each identifier $p \in \mathcal{P}$ to a primitive event stream containing instances with identifier p.

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

3.2 Composite events

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

Definition 6. If $A \in \mathcal{P}$, then A is an event expression. If A and B are event expressions, and $\tau \in \mathcal{T}$, then $A \vee B$, $A + B$, $A - B$, $A;B$ and A_τ are event expressions.

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

Definition 7. An event instance is a union of n primitive event instances, where $0 < n$.

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

Definition 8. For an event instance a we define

$$\begin{aligned} \mathrm{start}(a) &= \min(\{\tau \mid \langle p, v, \tau\rangle \in a\}) \\ \mathrm{end}(a) &= \max(\{\tau \mid \langle p, v, \tau\rangle \in a\}) \end{aligned}$$

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

Example 4. Let $a = \{\ \langle\mathsf{T}, 12, 2\rangle,\ \langle\mathsf{P}, \text{low}, 4\rangle,\ \langle\mathsf{T}, 8, 5\rangle\ \}$. Then a is an event instance, with $\mathrm{start}(a) = 2$ and $\mathrm{end}(a) = 5$.

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

Definition 9. An event stream is a set of event instances.

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

3.3 Semantics

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

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

Definition 10. For event streams S and T, and for $\tau \in \mathcal{T}$, define:

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

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

Definition 11. The meaning of an event expression for a given interpretation $\mathcal{I}$ is defined as follows:

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

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

These definitions result in an algebra with simple semantics and intuitive algebraic properties, but it is difficult to implement with limited resources. To deal with resource limitations, we define a formal restriction policy, and require only that an implementation should compute a valid restriction of the event stream specified by the algebra semantics.

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

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

Definition 12. For two event streams, S and S', $rem(S, S')$ holds if the following conditions hold:

1. $S' \subseteq S$
2. $\forall s\ (s\in S\ \Rightarrow\ \exists s'(s'\in S' \wedge \text{start}(s)\leq\text{start}(s') \wedge \text{end}(s)=\text{end}(s')))$
3. $\forall s, s'\ ((s\in S' \wedge s'\in S' \wedge \text{end}(s)=\text{end}(s'))\ \Rightarrow\ s=s')$

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

4. PROPERTIES

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

Definition 13. For event expressions A and B we define $A \equiv B$ to hold if $[\![A]\!]^{\mathcal{I}} = [\![B]\!]^{\mathcal{I}}$ for any interpretation $\mathcal{I}$.

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

THEOREM 1. *If $A \equiv A'$, $B \equiv B'$ and $\tau \in \mathcal{T}$, then we have $A\vee B \equiv A'\vee B'$, $A+B \equiv A'+B'$, $A;B \equiv A';B'$, $A-B \equiv A'-B'$ and $A_\tau \equiv A'_\tau$.*

PROOF. This follows in a straightforward way from Definitions 10 and 13. $\square$

Now, the laws can be formulated. For a more extensive set of laws, and formal proofs, the reader is refered to [3].

THEOREM 2. *For event expressions A, B and C, the following laws hold.*

$$\begin{array}{rcl} A\vee A & \equiv & A \\ A\vee B & \equiv & B\vee A \\ A+B & \equiv & B+A \\ A\vee(B\vee C) & \equiv & (A\vee B)\vee C \\ A+(B+C) & \equiv & (A+B)+C \\ A;(B;C) & \equiv & (A;B);C \\ (A\vee B)+C & \equiv & (A+C)\vee(B+C) \\ (A\vee B);C & \equiv & (A;C)\vee(B;C) \\ A;(B\vee C) & \equiv & (A;B)\vee(A;C) \\ (A-B)-C & \equiv & A-(B\vee C) \\ (A\vee B)-C & \equiv & (A-C)\vee(B-C) \end{array}$$

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

THEOREM 3. *For event expressions A and B, and $\tau\in\mathcal{T}$, the following laws hold.*

$$\begin{array}{rcl} A & \equiv & A_\tau \quad \text{if } A \in \mathcal{P} \\ (A_\tau)_{\tau'} & \equiv & A_{\min(\tau,\tau')} \\ (A\vee B)_\tau & \equiv & (A_\tau)\vee(B_\tau) \\ (A+B)_\tau & \equiv & (A_\tau+B)_\tau \\ (A-B)_\tau & \equiv & (A_\tau)-B \\ (A-B)_\tau & \equiv & (A-(B_\tau))_\tau \\ (A;B)_\tau & \equiv & (A_\tau;B)_\tau \\ (A;B)_\tau & \equiv & (A;B_\tau)_\tau \end{array}$$

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

THEOREM 4. *For event expressions A and A' such that $A \equiv A'$, if $rem([\![A]\!], S)$ holds then $rem([\![A']\!], S)$ holds as well.*

Thus, $A \equiv A'$ ensures that the result of an implementation detecting A is always a valid result for A'. As long as reasoning is based on the algebra semantics and the restriction policy, and not on the details of a particular detection algorithm, it will be equally valid for equivalent expressions.

To further investigate the relation between equivalent expressions when restriction is applied, notice that the restriction policy implies that detected event streams for equivalent expressions always contain instances with corresponding start and end times. This means that the part of the system that responds to the detected event occurrences is notified at the same time for equivalent expressions, but possibly with different values attached to the detected occurrences. Formally, we express this as follows.

Definition 14. For event streams S and T, define $S \cong T$ to hold if the following holds:

$$\{\langle \text{start}(s), \text{end}(s)\rangle \mid s \in S\} \;=\; \{\langle \text{start}(t), \text{end}(t)\rangle \mid t \in T\}$$

Trivially, $\cong$ is an equivalence relation.

THEOREM 5. *For event streams S, T and T', if $rem(S,T)$ and $rem(S,T')$ holds then $T \cong T'$*

PROOF. Take any $t \in T$. Then, since $T \subseteq S$, $t \in S$. By the second condition in the definition of *rem*, there 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 $\text{start}(t') = \text{start}(t)$. So, for any $t \in T$ there is a $t' \in T'$ with the same start and end time. Trivially, the opposite holds as well. $\square$

COROLLARY 1. *If $A \equiv A'$, $rem([\![A]\!], T)$ and $rem([\![A']\!], T')$, then $T \cong T'$.*

Thus, $A \equiv A'$ ensures that for any implementation consistent with the restriction policy, the instances found when detecting A and A' have the same start and end times. This means that the part of the system that responds to the detected event occurrences is notified at the same time for equivalent expressions, but possibly with different values attached to the detected occurrences.

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

THEOREM 6. *If $rem(S,S')$ and $rem(T,T')$ holds, than for any event stream U and $\tau \in \mathcal{T}$ the following implications hold:*

$$\begin{array}{lcl}
rem(\text{dis}(S',T'),\ U) & \Rightarrow & rem(\text{dis}(S,T),\ U) \\
rem(\text{con}(S',T'),\ U) & \Rightarrow & rem(\text{con}(S,T),\ U) \\
rem(\text{neg}(S',T'),\ U) & \Rightarrow & rem(\text{neg}(S,T),\ U) \\
rem(\text{seq}(S',T'),\ U) & \Rightarrow & rem(\text{seq}(S,T),\ U) \\
rem(\text{tim}(S',\tau),\ U) & \Rightarrow & rem(\text{tim}(S,\tau),\ U)
\end{array}$$

PROOF. We present the proof for negation, and refer to [3] for the full proof. Assume $rem(\text{neg}(S',T'), U)$. For any $u \in U$ we have $u \in \text{neg}(S',T')$ 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 $\text{start}(u) \leq \text{start}(t)$ and $\text{end}(t) \leq \text{end}(u)$, then there must exist some $t' \in T'$ with $\text{start}(t) \leq \text{start}(t')$ and $\text{end}(t') = \text{end}(t)$ which contradicts the fact that $u \in \text{neg}(S',T')$. Since no such t can exist, we have $u \in \text{neg}(S,T)$ and thus $U \subseteq \text{neg}(S,T)$, satisfying the first condition in the definition of *rem*.

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

Finally, $rem(\text{neg}(S',T'), U)$ ensures that all instances in U have different end times. Together, this shows that the three constraints of Definition 12 are satisfied, and thus $rem(\text{neg}(S,T), U)$ holds. $\square$

5. AN EVENT DETECTION ALGORITHM

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

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

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

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

5.1 Algorithm correctness

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

Definition 15. For $1 \leq i \leq m$, define

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

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

Due to space limitations, we only give an informal description for each operator, and refer to [3] for formal proofs.

```
for $i$ from 1 to $m$
    if $E^i \in \mathcal{P}$ then
        $a_i$ := the current instance of $E^i$, or $\langle\rangle$ if
            there is no current instance.
    if $E^i = E^j \vee E^k$ then
        if start$(a_j) \leq$ start$(a_k)$
            then $a_i := a_k$
            else $a_i := a_j$
    if $E^i = E^j + E^k$ then
        if start$(l_i) <$ start$(a_j)$ then $l_i := a_j$
        if start$(r_i) <$ start$(a_k)$ then $r_i := a_k$
        if $l_i = \langle\rangle$ or $r_i = \langle\rangle$ or ($a_j = \langle\rangle$ and $a_k = \langle\rangle$)
            then $a_i := \langle\rangle$
            else if start$(a_k) \leq$ start$(a_j)$
                then $a_i := a_j \cup r_i$
                else $a_i := l_i \cup a_k$
    if $E^i = E^j - E^k$ then
        if $t_i <$ start$(a_k)$ then $t_i :=$ start$(a_k)$
        if $t_i <$ start$(a_j)$ then $a_i := a_j$ else $a_i := \langle\rangle$
    if $E^i = E^j ; E^k$ then
        $a_i := \langle\rangle$
        if $a_k \neq \langle\rangle$ then
            foreach $e$ in $q_i$
                if end$(e) <$ start$(a_k)$ and start$(a_i) <$ start$(e)$
                    then $a_i := e$
            if $a_i \neq \langle\rangle$ then $a_i := a_k \cup a_i$
        if $t_i <$ start$(a_j)$ then
            $q_i := q_i \cup \{a_j\}$
            $t_i :=$ start$(a_j)$
    if $E^i = (E^j)_\tau$ then
        if end$(a_j) -$ start$(a_j) \leq \tau$
            then $a_i := a_j$
            else $a_i := \langle\rangle$
```

Figure 2: Algorithm for detecting event expression E. Initially, $t_i = -1$, $l_i = r_i = \langle\rangle$ and $q_i = \emptyset$ for $1 \leq i \leq m$.

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

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

The fact that the implementation of the disjunction operator corresponds to the declarative semantics with restriction, with respect to the instances that are detected for the subexpressions, can be formulated as *rem*(dis$(\mathcal{A}(j), \mathcal{A}(k)), \mathcal{A}(i))$.

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

For conjunctions, it is necessary to store the instance with maximum start time so far from each of the two subexpressions. At the start of time instant τ, l_i is an element in $\{e \mid e \in \mathcal{A}(j) \wedge \text{end}(e) < \tau\}$ with maximum start time, or $\langle\rangle$ if that set is empty. The corresponding property holds for r_i and $\mathcal{A}(k)$.

The first two conditionals in the conjunction part of the algorithm correctly update the state for the next time instant. The new state is then used to construct the current output instance for the conjunction, if there is one.

```
$a_1$ := the current instance of T, or $\langle\rangle$ if there is none.
$a_2$ := the current instance of P, or $\langle\rangle$ if there is none.
if start$(a_1) \leq$ start$(a_2)$ then $a_3 := a_2$ else $a_3 := a_1$
$a_4$ := the current instance of B, or $\langle\rangle$ if there is none.
if $t_5 <$ start$(a_4)$ then $t_5 :=$ start$(a_4)$
if $t_5 <$ start$(a_3)$ then $a_5 := a_3$ else $a_5 := \langle\rangle$
```

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

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

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

At the start of time instant τ, t_i should be the maximum start time of the elements in $\{e \mid e \in \mathcal{A}(k) \wedge \text{end}(e) < \tau\}$, or -1 if this set is empty. The first conditional updates the state so that it is valid for the next time instant. Then, the updated t_i variable is used to check if the current instance of E^j is invalidated or not.

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

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

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

The first conditional selects the best match for the current E^k instance from the instances stored in q_i, and creates the output instance if a matching instance is found. Then, in the second conditional, the state is updated. To ensure that q_i does not contain fully overlapping instances, the E^j instance is checked against t_i before it is added.

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

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

Putting it all together

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

THEOREM 7. *For any $1 \leq i \leq m$, rem$([\![E^i]\!], \mathcal{A}(i))$ holds.*

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

```
        $a_i := \langle\rangle$
        if $a_k \neq \langle\rangle$ then
            foreach $e$ in $q_i$
                if $\mathrm{end}(e) < \mathrm{start}(a_k)$ and $\mathrm{start}(a_i) < \mathrm{start}(e)$
                    then $a_i := e$
            if $a_i = \langle\rangle$ then $a_i := l_i$
            if $a_i \neq \langle\rangle$ then $a_i := a_k \cup a_i$
        if $t_i < \mathrm{start}(a_j)$ then $q_i := q_i \cup \{a_j\}$;  $t_i := \mathrm{start}(a_j)$
        foreach $e$ in $q_i$
            if $\mathrm{end}(e) \leq \tau^c - \tau'$ then $q_i := q_i - \{e\}$;  $l_i := e$
```

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

Next, assume that for some i, $\mathit{rem}([\![E^x]\!], \mathcal{A}(x))$ holds for any $1 \leq x < i$. If $E^i = E^j \vee E^k$, then according to the discussion above we have $\mathit{rem}(\mathrm{dis}(\mathcal{A}(j), \mathcal{A}(k)), \mathcal{A}(i))$. Since the subexpressions are numbered bottom-up, we have $j < i$ and $k < i$, so by assumption $\mathit{rem}([\![E^j]\!], \mathcal{A}(j))$ and $\mathit{rem}([\![E^k]\!], \mathcal{A}(k))$ holds. According to Theorem 6, $\mathit{rem}(\mathrm{dis}([\![E^j]\!], [\![E^k]\!]), \mathcal{A}(i))$ holds, which means that $\mathit{rem}([\![E^i]\!], \mathcal{A}(i))$ holds. A similar proof can be constructed for each of the operators. By induction, the theorem holds. $\square$

5.2 Memory complexity

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

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

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

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

For large values of τ, for example in systems with a fine granularity timebase, this resource bound might not be sufficient in practice. Also, for $A;B$ the bound is a large over-approximation unless A occurs very frequently. If we have information about the minimum separation time of primitive events (i.e., the minimum time between two consecutive occurrences of the same event), more precise worst case memory estimates can be derived [3].

$$
\begin{aligned}
&\mathrm{transform}(A, \tau) = \langle A, 0 \rangle \qquad \text{if } A \in \mathcal{P} \\
&\mathrm{transform}(A \vee B, \tau) = \langle A' \vee B', \max(\tau_a, \tau_b) \rangle \\
&\qquad \text{where } \langle A', \tau_a \rangle = \mathrm{transform}(A, \tau) \\
&\qquad\qquad\quad \langle B', \tau_b \rangle = \mathrm{transform}(B, \tau) \\
&\mathrm{transform}(A + B, \tau) = \langle A' + B', \infty \rangle \\
&\qquad \text{where } \langle A', \tau_a \rangle = \mathrm{transform}(A, \tau) \\
&\qquad\qquad\quad \langle B', \tau_b \rangle = \mathrm{transform}(B, \tau) \\
&\mathrm{transform}(A - B, \tau) = \langle A' - B', \tau_a \rangle \\
&\qquad \text{where } \langle A', \tau_a \rangle = \mathrm{transform}(A, \tau) \\
&\qquad\qquad\quad \langle B', \tau_b \rangle = \mathrm{transform}(B, \min(\tau_a, \tau)) \\
&\mathrm{transform}(A;B, \tau) = \begin{cases} \langle A';_{\tau_b} B', \infty \rangle & \text{if } \tau_b \leq \tau \\ \langle A';_{\tau} (B'_\tau), \infty \rangle & \text{if } \tau < \tau_b \end{cases} \\
&\qquad \text{where } \langle A', \tau_a \rangle = \mathrm{transform}(A, \tau) \\
&\qquad\qquad\quad \langle B', \tau_b \rangle = \mathrm{transform}(B, \tau) \\
&\mathrm{transform}(A_{\tau'}, \tau) = \begin{cases} \langle A', \tau_a \rangle & \text{if } \tau_a \leq \tau'' \\ \langle A'_{\tau''}, \tau'' \rangle & \text{if } \tau'' < \tau_a \end{cases} \\
&\qquad \text{where } \langle A', \tau_a \rangle = \mathrm{transform}(A, \tau'') \\
&\qquad\qquad\qquad\quad \tau'' = \min(\tau, \tau')
\end{aligned}
$$

Figure 5: The transformation function

5.3 Time complexity

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

Due to the particular characteristics of q_i, a more elaborate implementation is possible, where the best match is found in logarithmic time and updates are only bounded by the m factor for variable-sized instances (for details, see [3]). Using this implementation, the total complexity is $O(mn'')$, where $n'' = \max(m, \log n)$.

6. TRANSFORMATION ALGORITHM

This section describes how event expressions can be automatically transformed into equivalent expressions that allow a more efficient detection. The transformation algorithm is based on the algebraic laws describing how temporal restrictions can be propagated through an expression, presented in Theorem 3.

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

The transformation algorithm is based on a recursive function that takes an expression and a time as input, and returns the transformed expression and a time. This function is defined in Figure 5. The input time represents a temporal restriction that can be applied to the expression without changing the meaning of the expression as a whole. The returned time represents a temporal restriction that can be applied to the transformed expression without changing its meaning.

The meaning of an expression is unchanged by the algorithm, and sequences are labeled correctly, as specified by the following theorem. For the proof, the reader is refered to [3].

THEOREM 8. *If* $\text{transform}(E, \infty) = \langle E', \tau' \rangle$ *then* $E \equiv E'$. *Also, all sequences in* E' *are on the labeled form* $A;_\tau B$*, where* $B \equiv B_\tau$ *holds.*

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

Example 5. We have $\text{transform}((\mathsf{B};\mathsf{B})_2 - (\mathsf{P};(\mathsf{P}+\mathsf{T})),\ \infty) = \langle(\mathsf{B};_0\mathsf{B})_2 - (\mathsf{P};_2(\mathsf{P}+\mathsf{T})_2),\ 2\rangle$ which means that this expression can be detected with limited memory. Note that the temporal restriction in the left subexpression of the negation has been propagated to the right subexpression, making it detectable with limited memory.

7. CONCLUSIONS AND FUTURE WORK

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

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

Our ongoing work includes investigating how to combine the algebra with languages that specifically target reactive systems, in particular Esterel [2], FRP [13, 15] and Timber [5]. We are also considering extending the algebra with a delay operator to allow the definition of timeout events, and adding support for manipulating the values associated with event occurrences.

8. REFERENCES

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

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

[3] J. Carlson. An intuitive and resource-efficient event detection algebra. Licentiate thesis No. 29, June 2004. Mälardalen University, Sweden.

[4] J. Carlson and B. Lisper. An interval-based algebra for restricted event detection. In *First Int. Workshop on Formal Modeling and Analysis of Timed Systems (FORMATS 2003)*, volume 2791 of *Lecture Notes in Computer Science.* Springer-Verlag, Sept. 2003.

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

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

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

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

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

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

[11] G. Liu, A. Mok, and P. Konana. A unified approach for specifying timing constraints and composite events in active real-time database systems. In *4th IEEE Real-Time Technology and Applications Symposium (RTAS '98)*, pages 199–209. IEEE, June 1998.

[12] J. Mellin. *Resource-Predictable and Efficient Monitoring of Events.* PhD thesis, Department of Computer Science, University of Skövde, June 2004.

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

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

[15] Z. Wan, W. Taha, and P. Hudak. Event-driven FRP. In *Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages*, volume 2257 of *Lecture Notes in Computer Science*, 2002.