Expressiveness Issues and Decision Problems for Active Database Event Queries

James Bailey and Szabolcs Mikulás*

Department of Computer Science, Birkbeck College, Univ. of London
Malet Street, London WC1E 7HX, UK. {james,szabolcs}@dcs.bbk.ac.uk

Abstract. A key facility of active database management systems is their ability to detect and react to the occurrence of events. Such events can be either atomic in nature, or specified using an event algebra to form complex events. An important role of an event algebra is to define the semantics of when events become invalid (event consumption). In this paper, we examine a simple event algebra and provide a logical framework for specification of various consumption policies. We then study the problems of equivalence and implication, identifying a powerful class of complex events for which equivalence is decidable. We then demonstrate how extensions of this class lead to undecidability.

1 Introduction

First we briefly introduce the context of our work, list our contributions and survey related works.

1.1 Events in Active Databases

Active databases provide the functionality of traditional databases and additionally are capable of reacting automatically to state changes without user intervention. This is achieved by means of Event-Condition-Action (ECA) rules of the form *on* event *if* condition *do* action. The event part of an ECA rule specifies the points in time when the rule should become triggered and is the focus of this paper. Events are of two kinds: primitive events correspond to atomic, detectable occurrences, while complex events are combinations of simple events specified using an *event algebra*. Detecting the occurrence of a complex event corresponds to the problem of evaluating an *event query* (specified using the event algebra) over a history of all previously occurring primitive events.[1] A notable characteristic of active database prototypes has been the rich and expressive event algebras developed [11]. An interesting feature of such algebras is the notion of *event consumption policies*. These are used to specify when certain instances of an event should no longer be considered in the detection of other

* Supported by EPSRC No. GR/L26872.

[1] In this paper, we will use the terms complex event detection and (complex) event query evaluation over a history synonymously.

events depending on it. Previous work has predominantly dealt with the definition of policies for a range of applications, without focusing on their formal foundations. Indeed, there is not even widespread agreement on which policies should be standard features of an event algebra. In contrast, one of the purposes of our work is to give a framework for specifying event consumption policies and investigate what consequences their inclusion has for the expressiveness of the event query language as a whole.

In addition to expressiveness, another important question is to consider what kinds of analysis are possible for different classes of event queries. For example, the histories over which event queries may be evaluated can be very large and it is important to avoid redundant evaluation. In fact, efficient event detection has been identified as being a major challenge in improving active DBMS system performance ([6]). One possible optimisation involves being able to reason about event query equivalence and implication. Given event queries q_1 and q_2, if $q_1 \Leftrightarrow q_2$, then only one of $\{q_1, q_2\}$ needs be evaluated over the event history (whichever is less expensive, according to some cost model). Alternatively, if $q_1 \Rightarrow q_2$, then one need not evaluate q_2 whenever q_1 is true. Being able to decide these questions is therefore a useful property for an event query language to possess and the second half of this paper provides (un)decidability results for a core event language and a selection of consumption policies.

1.2 Contributions

We make two main contributions in this paper. The first is a formal procedure for the specification of event consumption which is then useful for comparison of consumption policies. The second contribution is the presentation of decidability and undecidability results for implication and equivalence of event queries. We define a powerful query class for which equivalence is decidable, yet implication is undecidable. Our results also highlight the important role that event consumption has in determining whether an effective analysis is possible. To our knowledge, this is the first paper to consider decision questions for event queries making use of different consumption policies.

1.3 Related Work

A number of active database prototypes have been built providing sophisticated event algebras (e.g. SNOOP, REACH, NAOS, CHIMERA [11]), but there are not associated results on the expressiveness of these algebras which could be used as the basis for (global) reasoning about implication and equivalence of event queries. The event language of the system ODE [5], however, has been shown to have the power of regular expressions, but event consumption policies are not an explicit feature of the language. [10] proposes the use of Datalog_{1S} for expressing complex event queries. The advantage being that it has a well-defined formal semantics. Such a language is rather more powerful than the algebras we consider in this paper, however, and indeed equivalence of Datalog_{1S} expressions can easily be shown to be undecidable. Similar considerations apply to work on modelling complex events using the Kowalski–Sergot event calculus [3] or coloured Petri nets [4] as used in SAMOS.

An early work which recognised the importance of event consumption is [1], where a variety of so-called parameter contexts were proposed for matching and consuming events. Work in [15] provides a meta-model for classifying a number of properties of complex event formalisms.

Equivalence and implication are of course important problems elsewhere in database theory, e.g., for conjunctive queries. It is not obvious how to use these results for reasoning about event queries, since the queries considered usually lack one of the following features: an ordering on the underlying domain, negation, or the ability to mimic event consumption. Work on temporal logics, however, is directly relevant for event reasoning, and we discuss and make use of these results in the following section.

Other work on temporal aspects of active databases includes [2] and [13], where the focus is on evaluation and expressiveness of temporal conditions. A characteristic that distinguishes conditions from events, however, is that they do not have associated consumption policies. Event languages are also used in other database areas, such as multimedia, where they are used in specifying interaction with documents [9].

2 Semantics for Complex Events

We assume the following about events in the spirit of [15]: a) The time domain is the set of natural numbers. b) Primitive events are detected by the system at points in time — that is, we can treat primitive events as primitive symbols interpreted in time. c) Event types are independent and they can be freely used as components to generate complex events — i.e., we can treat them as propositional variables that we can use for building compound formulas. d) The occurrence time of the terminator of a complex event is the occurrence time of the complex event, i.e., an event occurs when the last component of the event has occurred. e) Events can occur simultaneously.

Note that we model complex events as boolean queries over the history of all previously occurring primitive events. An event query is true over a history iff it is true at the last point in the history (of course it may well be true for prefixes of the history also).

Remark 1. We give two preliminary observations concerning our approach.

a) In practice, primitive or complex events could have other information associated with them (e.g., parameters or variables describing why they occurred)[2]. We ignore such a refinement in this paper, instead concentrating on the problem of recognising whether an event has occurred, as opposed to explaining why it occurred.

[2] If the domain of such parameters is finite, then we could of course model them by defining extra primitive events.

b) We are considering the definition of event queries and not techniques for their evaluation. In practice, incremental methods may be appropriate, but such evaluation methods are an orthogonal issue to the problems of expressiveness and analysis that we address.

2.1 Operations on Events

Next we review the most often used operations on events. We also propose a concise and formal way to express events occurring in time by using temporal logic over natural numbers.

We will model events occurring in active databases in time. Let $\mathfrak{N} = (\mathbb{N}, <, I)$ be a structure such that $\mathbb{N} = \{0, 1, \dots\}$ is the set of natural numbers, $<$ is their usual ordering according to magnitude, and I is an interpretation function associating a subset $I(p)$ of $\mathbb{N}$ to every primitive event p (i.e., when it occurred).

Basic operations For given events e and e', we first consider the following basic operations:

- *sequence operation* $e \,;\, e'$: events e and e' should occur in this order;
- *simultaneous operation* $e \parallel e'$: the events occur simultaneously;
- *conjunction operation* $e \sqcap e'$: both occur but it can happen in any order;
- *disjunction operation* $e \sqcup e'$: at least one of them occurs;
- *negation operation* $\sim e$: e does not occur (at a given time point).

Using the semantics above, it is straightforward to define when an event e occurs at a time-point $t \in \mathbb{N}$ — notation: $\mathfrak{N}, t \models e$. Note that $e \sqcap e'$ can be defined in terms of $\sqcup$, $\parallel$ and ;: $(e \parallel e') \sqcup (e \,;\, e') \sqcup (e' \,;\, e)$.

We will see shortly how to express the above events using temporal logic, and we will also extend this translation to more complex event definitions as well.

2.2 Temporal Logic

Next we briefly recall the basics of temporal logic.[3] The language of our propositional temporal logic contains the usual propositional connectives *conjunction* $\wedge$ and *negation* $\neg$ and the binary temporal connectives *until* $\mathcal{U}$ and *since* $\mathcal{S}$. We also assume that a countable set P of propositional variables is at hand.

Other propositional connectives (disjunction $\vee$, implication $\rightarrow$, true $\top$, false $\bot$) are defined in the usual way. We define the unary temporal connectives *future* $\mathcal{F}$ and *past* $\mathcal{P}$:

$$\mathcal{F}\varphi \stackrel{\text{def}}{=} \mathcal{U}(\varphi, \top) \quad \text{and} \quad \mathcal{P}\varphi \stackrel{\text{def}}{=} \mathcal{S}(\varphi, \top).$$

As usual, $\mathcal{G}\varphi$ denotes $\neg\mathcal{F}\neg\varphi$ and $\mathcal{H}\varphi$ stands for $\neg\mathcal{P}\neg\varphi$, while $\Diamond\varphi$ is a shorthand for $\mathcal{P}\varphi \vee \mathcal{F}\varphi \vee \varphi$, and $\Box\varphi \stackrel{\text{def}}{=} \neg\Diamond\neg\varphi$. We will use the following definitions as well:

[3] We will use only past operators for modelling events in this paper, but we present a general framework that will enable us to include, e.g., rules as well.

$\triangleright\varphi \stackrel{\text{def}}{=} \mathcal{U}(\varphi, \bot)$ for *next*, $\triangleleft\varphi \stackrel{\text{def}}{=} \mathcal{S}(\varphi, \bot)$ for *previous*, and $\odot\varphi \stackrel{\text{def}}{=} \Diamond(\varphi \wedge \neg\mathcal{P}\top)$ for *beginning*.

In some applications, e.g., for expressing periodic events, it will be useful to consider a temporal language with an additional fixed point operation $\mu p.\varphi$. It is defined in the usual way: roughly speaking, if the free occurrences of the propositional symbol p are in the scope of past temporal operators in φ, then we can form the formula $\mu p.\varphi$. Since at any time point the past is finite (we have natural numbers flow of time), this guarantees the existence of a fixed point; see [7] for more details.

We are going to interpret temporal formulas over flows of time on the natural numbers. In more detail, let $\mathbb{N} = 0, 1, \ldots, n, \ldots$ be the set of natural numbers and $<$ their usual ordering (according to magnitude). Let v be an evaluation of the propositional variables in P, i.e., $v : P \to \mathcal{P}(\mathbb{N})$. Then truth of formulas in the model $\mathfrak{N} = (\mathbb{N}, <, v)$ is defined in the usual way; the case of the temporal connectives is as follows: for every $n \in \mathbb{N}$,

$$\begin{aligned}
\mathfrak{N}, n \Vdash \mathcal{U}(\varphi, \psi) \Longleftrightarrow{} & \text{ for some } m > n, \mathfrak{N}, m \Vdash \varphi \\
& \text{ and for all } l \text{ with } n < l < m, \mathfrak{N}, l \Vdash \psi \\
\mathfrak{N}, n \Vdash \mathcal{S}(\varphi, \psi) \Longleftrightarrow{} & \text{ for some } m < n, \mathfrak{N}, m \Vdash \varphi \\
& \text{ and for all } l \text{ with } m < l < n, \mathfrak{N}, l \Vdash \psi.
\end{aligned}$$

Finally, the interpretation of fixed point formulas is as follows.

$$(\mathbb{N}, <, v), n \Vdash \mu p.\varphi \quad \text{iff} \quad n \in w(p)$$

for that unique evaluation w of the propositional variables that satisfies

- for every atom q distinct from p, $w(q) = v(q)$,
- $n \in w(p)$ iff $(\mathbb{N}, <, w), n \Vdash \varphi$.

Theorem 1. *The complexity of the decision problem for temporal logic of $\mathcal{U}$ and $\mathcal{S}$ over natural numbers flow of time is* PSPACE*, [12]. This remains true even if we include the fixed point operator, [7].*

2.3 Complex Event Definitions

First we give the translations of the basic operations on events into temporal logic:

- $e_{tr} \equiv e$ for primitive event e;
- $(e\, ;e')_{tr} \equiv e'_{tr} \wedge \mathcal{P}e_{tr}$;
- $(e \parallel e')_{tr} \equiv e_{tr} \wedge e'_{tr}$;
- $(e \sqcap e')_{tr} \equiv (e_{tr} \wedge e'_{tr}) \vee (e_{tr} \wedge \mathcal{P}e'_{tr}) \vee (e'_{tr} \wedge \mathcal{P}e_{tr})$;
- $(e \sqcup e')_{tr} \equiv e_{tr} \vee e'_{tr}$;
- $(\sim e)_{tr} \equiv \neg e_{tr}$.

3 Consumption Policies

Event consumption policies are used to specify when certain instances of an event should no longer be considered in the detection of other events that depend on it. There are many ways in which event instances might be consumed (or not) when composite events are detected. In this section, we recall some representative policies for consuming events, similar to those of [1]. We propose a formal logic-based definition for consumption policies that will enable us to look at the problem of the complexity of deciding if two events together with consumption policies are equivalent.

Several consumption policies have been considered for the sequence operation on events. First we give a general definition, then look at particular policies and express them in the general framework. In the next section, we will consider the complexity of deciding equivalence and implication between event queries using consumption policies.

Definition 1. *Let $\mathfrak{N} = (\mathbb{N}, <, I)$ be a model with natural numbers universe $\mathbb{N}$ together with the usual ordering $<$ and an interpretation function I that associates a subset $I(p)$ of $\mathbb{N}$ to every primitive event p. Let e and e' be events and $R \subseteq \mathbb{N} \times \mathbb{N}$ be a binary relation that may depend on e and e'. Then we define the sequence operation $;_R$ as follows:*

$$\mathfrak{N}, t \models (e ;_R e') \iff \mathfrak{N}, t \models e' \text{ and } \mathfrak{N}, t' \models e \text{ for some } t' < t \text{ such that } R(t', t).$$

That is, $e ;_R e'$ holds at a time-point t iff $e ; e'$ occurs at t and the witness t' for e is related to t by R.

Note that by this definition we have a recursive way of defining more complex events under consumption policies. For instance, given two relations R and S, the definition of the event $(p ;_R q) ;_S r$ is as follows. Given a model $\mathfrak{N}$, first apply the definition of $;_R$ to determine for each $t \in \mathbb{N}$ if $p ;_R q$ occurs at t; then apply the definition of $;_S$ to $e ;_S r$ where e is the event $p ;_R q$. In other words, we compute a composite event by first computing the participating events and then, treating them as atomic events, the composite event. We also note that the above definition is in the style of modal logic, where the extra-boolean operations are evaluated according to accessibility relations. More generally, our definition of event consumption is compositional in the sense that the meaning (or truth) of a formula is determined by the meaning of its subformulas (plus the structure of the model).

Particular consumption policies are defined by imposing certain conditions on the relation R. For instance, the sequence operation $;$ can be treated as an operation with the most permissive consumption policy. In this case, we can take R as the universal binary relation, i.e., there is no further restriction on the p-witness t' for $\mathfrak{N}, t \models p ; q$ other than $t' < t$. Furthermore, as we will see below for the most-recent and cumulative consumption policies, there are cases when R is essentially a unary predicate. This suggests a definition of a subclass CP_1 of consumption policies when R can be substituted by a first-order formula using only unary predicates and the ordering $<$ in Definition 1. We already mentioned that the temporal logic of $\mathcal{U}$ and $\mathcal{S}$ is expressively complete w.r.t. first-order logic over natural numbers. Then for every operation $;_R \in CP_1$, there is an equivalent temporal logic formula φ_R with parameters:

$$\mathfrak{N}, t \models p ;_R q \iff \mathfrak{N}, t \Vdash \varphi_R(p, q). \tag{1}$$

We can further strengthen this result by allowing definitions of R using monadic second-order logic $S1S$. Let CP_{S1S} be the resulting class of consumption policies. Again we have the equivalence result that the temporal logic of $\mathcal{U}$, $\mathcal{S}$ and fixed point is expressively complete w.r.t. $S1S$ over natural numbers, [7]. Thus the above equivalence 1 holds for CP_{S1S} and fixed point temporal logic.

We will demonstrate the idea of consumption policies on the example of the composite event e which is the sequential composition of the (primitive) events p and q. We will use the following example, see Figure 1. Let us assume that we are at time point 11 and the events p and q happened as follows;

$$\begin{aligned} I(p) &= \{0, 1, 2, 5, 9, 10\} \\ I(q) &= \{3, 4, 6, 7, 8, 11\}. \end{aligned}$$

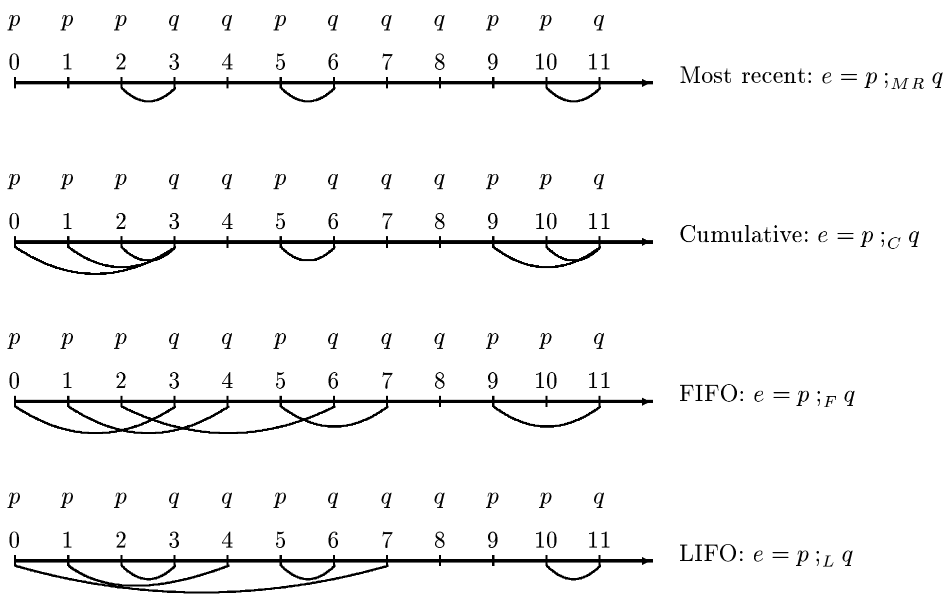

Fig. 1. Consumption policies

Most recent In this context, only the most recent occurrence of the initiator (the first component of the event e) p is used. After detecting e, that occurrence of p which initiated e is "consumed", i.e., it cannot initiate other occurrence of e. Also the terminator (the final component of the event e) is consumed, i.e., it cannot be the terminator of another occurrence of e. Such a policy is useful for situations where events occur rapidly and successive occurrences only refine the previous value (e.g., sensor applications).

In the example above, e is first detected at 3 when e is initiated by the occurrence of p at 2 and terminated by the occurrence of q at 3; we will denote this by $(e,3) = (p,2;q,3)$. Then other occurrences of e are as follows: $(e,6) = (p,5;q,6)$ and $(e,11) = (p,10;q,11)$.

This consumption policy can be defined by the following relation R in the above definition:

$$\begin{aligned} R(t,t') \iff & \mathfrak{N},t \models q,\ \mathfrak{N},t' \models p,\ t' < t, \\ & \text{and for each } t'',\ \text{if } t' < t'' < t \text{ then } \mathfrak{N},t'' \not\models p \text{ and } \mathfrak{N},t'' \not\models q. \end{aligned}$$

That is, R is the partial function that associates consecutive p- and q-points. Note that in the above definition of R, there was no binary relation apart from $<$. Thus there is a temporal formula expressing this consumption policy:

$$\mathfrak{N},t \Vdash q \wedge \mathcal{S}(p, \neg p \wedge \neg q) \iff \mathfrak{N},t \models p \,;_R\, q.$$

We will use the notation $e;_{MR}e'$ for denoting the event $e;e'$ using the most-recent consumption policy.[4]

Cumulative In this version, all occurrences of the initiator are accumulated until the composite event is detected, and all these occurrences are consumed when the composite event is detected. Such a policy is useful where multiple occurrences of an event need to be grouped together (e.g. multiple deposits preceding a withdrawal). Using the above notation, we have $(e,3) = (p,0,1,2;q,3)$, $(e,6) = (p,5;q,6)$ and $(e,11) = (p,9,10;q,11)$. The definition of R is as follows:

$$\begin{aligned} R(t,t') \iff & \mathfrak{N},t \models q\ ,\ \mathfrak{N},t' \models p\ ,\ t' < t, \\ & \text{and for each } t'',\ \text{if } t' < t'' < t \text{ then } \mathfrak{N},t'' \not\models q. \end{aligned}$$

Again, we can express this by a temporal formula $q \wedge \mathcal{S}(p, \neg q)$. We will use the notation $e \,;_C\, e'$ for denoting the event $e \,;\, e'$ using the cumulative consumption policy.

Using the above definition of 'most recent' and 'cumulative' (or the corresponding temporal logic formulas), one easily shows the following.

Claim. The most-recent and cumulative consumption policies are equivalent. That is, for any given model $\mathfrak{N}$ and $t \in \mathbb{N}$, we have

$$\mathfrak{N},t \models e \,;_{MR}\, e' \iff \mathfrak{N},t \models e \,;_C\, e'.$$

[4] From now on, the subscript X in $;_X$ will stand for a particular consumption policy instead of the underlying binary relation.

FIFO [5] The event e is detected whenever q occurs and there is an "unconsumed" earlier occurrence of p, further the *earliest* such occurrence of p is the initiator of the event. An example use of this policy could be if p represented the availability of a service instance and q represented a request for such a service. We have $(e,3) = (p,0;q,3)$, $(e,4) = (p,1;q,4)$, $(e,6) = (p,2;q,6)$, $(e,7) = (p,5;q,7)$ and $(e,11) = (p,9;q,11)$.

The underlying relation R is defined in a recursive way in this case. For every $n \in \mathbb{N}$, let $\mathfrak{N}_n$ be the initial segment of $\mathfrak{N}$ defined by n: $\mathfrak{N}_n$ is the model $\mathfrak{N}$ relativized to $\mathbb{N}_n = \{t \in \mathbb{N} : t \leq n\}$, i.e., we restrict the universe and the interpretation of events to $\{t \in \mathbb{N} : t \leq n\}$. We define R_0 be the empty relation. Now let us assume that $R_n \subseteq \mathbb{N}_n \times \mathbb{N}_n$ has been already defined. R_{n+1} is defined as follows. If $\mathfrak{N}, n+1 \not\models q$, then $R_{n+1} = R_n$. If $\mathfrak{N}, n+1 \models q$, then we define $R_{n+1} = R_n \cup \{(m, n+1)\}$, where $m \leq n$ satisfies

$$\begin{aligned}&\mathfrak{N}, m \models p, \text{ for each } t \leq n,\ (m,t) \notin R_n,\\ &\text{and for each } t < m, \text{ if } \mathfrak{N}, t \models p \text{ then } R_n(t,t') \text{ for some } t' \leq n,\end{aligned}$$

provided such m exists; otherwise we define $R_{n+1} = R_n$. We let R be the union of the chain $(R_i : i \in \mathbb{N})$. Thus R is a partial injective function that associates every q-point x to the earliest p-point y such that $y < x$ and there is no $z < x$ with $R(y,z)$. Then $;_R$ defines the sequence operation under the FIFO consumption policy. We will use the notation $e ;_F e'$ for denoting the event $e ; e'$ using the FIFO consumption policy.

LIFO [6] This is a similar policy to the previous one, but the event e is initiated by the *last* "unconsumed" previous occurrence of p. In our example, we have the following occurrences: $(e,3) = (p,2;q,3)$, $(e,4) = (p,1;q,4)$, $(e,6) = (p,5;q,6)$, $(e,7) = (p,0;q,7)$ and $(e,11) = (p,10;q,11)$.

Again the corresponding relation R is defined recursively. We define R_0 be the empty relation. Now let us assume that $R_n \subseteq \mathbb{N}_n \times \mathbb{N}_n$ has been already defined. First assume that $\mathfrak{N}, n+1 \not\models q$ and in this case define $R_{n+1} = R_n$. If $\mathfrak{N}, n+1 \models q$, then $R_{n+1} = R_N \cup \{(m, n+1)\}$ where $m \leq n$ satisfies

$$\begin{aligned}&\mathfrak{N}, m \models p, \text{ for each } t \leq n,\ (m,t) \notin R_n,\\ &\text{for each } t, \text{ if } m < t < n+1 \text{ and } \mathfrak{N}, t \models p \text{ then } R_n(t,t') \text{ for some } t',\end{aligned}$$

provided such m exists; otherwise $R_{n+1} = R_n$. We let R be the union of the chain $(R_i : i \in \mathbb{N})$. Then $;_R$ defines the sequence operation under the FIFO consumption policy. We will use the notation $e ;_L e'$ for denoting the event $e ; e'$ using the LIFO consumption policy. Again, R satisfies the following condition: it is a partial function that maps every p-point x to the first q-point y such that $x < y$ and there is no $z < y$ with $R(x,z)$. Then it is not surprising that satisfiability of FIFO and LIFO events coincide.

Claim. The FIFO and LIFO consumption policies are equivalent.

[5] First in first out.

[6] Last in first out.

Remark 2. Although we have only looked at four examples, there are obviously (infinitely many) other consumption policies in the classes CP_1 and CP_{S1S}. The purpose of defining these two classes is as a classification mechanism and, as we shall see, CP_1 and CP_{S1S} represent classes of policies for which analysis is "easy".

Remark 3. The results of the next section imply that $;_L$ and $;_F$ are not definable in terms of CP_1 or CP_{S1S} operations. Intuitively, this is because their definition requires the relation R to be a "true" binary relation (one that can't be simulated by operations on unary relations).

4 Decidability of Events

In this section we look at the problem of deciding equivalence between events using consumption policies. We will consider several *event algebras* determined by various choices of operations. Let $\mathcal{E}_1$ be the event algebra in which events are generated by the following operations: sequence ; with any consumption policy from $CP_1 \cup CP_{S1S}$, $\|$, $\sqcap$, $\sqcup$, $\sim$. In the definition of $\mathcal{E}_2$, we allow any basic operation $\|$, $\sqcap$, $\sqcup$, $\sim$, plus the sequence operation under the most-recent, cumulative, FIFO and LIFO policies $;_I$ where $I \in \{MR, C, F, L\}$.[7] So, $\mathcal{E}_1$ is an event algebra that includes the basic operations plus the ability to define "well behaved" consumption policies. $\mathcal{E}_2$ is an event algebra including the same basic operations, but allowing the use of the more expressive policies $;_L$ and $;_F$.

Our aim is to find out the complexity of implication and equivalence. The *implication problem* is defined by:

given two events e and e', whether for every $\mathfrak{N}$ and $t \in \mathbb{N}$, $\mathfrak{N}, t \models e$ implies $\mathfrak{N}, t \models e'$.

We will write $e \Rightarrow e'$ if e' is implied by e. *Equivalence* of events, in symbols $e \Leftrightarrow e'$, is defined by mutual implication.

Theorem 2. *Let e and e' be two event queries written in the event algebra $\mathcal{E}_1$. The complexity of deciding implication and equivalence between e and e' is in* PSPACE.

Next we look at the problem of equivalence between queries that can use either the $;_F$ and $;_L$ operator (but not both). An additional restriction is that the chosen operator from $\{;_L, ;_F\}$ can be used no more than once. Such a restriction would seem realistic in practice, given the likely difficulty in understanding/writing event queries using $\{;_L, ;_F\}$ multiple times.

Theorem 3. *Let q_1 and q_2 be event queries of the event algebra $\mathcal{E}_2$ such that each uses one instance of either $;_F$ or $;_L$. Then the problem of whether $q_1 \Leftrightarrow q_2$ is decidable.*

[7] As future work, we plan to extend the definition of $\mathcal{E}_2$ to also allow any policy in $CP_1 \cup CP_{S1S}$.

Proof. We show how to translate each q_i ($i \in \{1,2\}$) into a deterministic one counter machine. Since equivalences of such machines is decidable (in exponential time of the size of the machines [14]), it follows that equivalence of q_1 and q_2 is decidable. The difficulty of the translation arises from ensuring that the constructed machine is deterministic.

A deterministic one counter machine (docm) is a deterministic pushdown automaton having a stack alphabet of only one symbol. In more detail,

$$M = (\Sigma, Acc, St, A, \delta)$$

where Σ is a finite set of states, $Acc \subseteq \Sigma$ is the set of accepting states, $St \in \Sigma$ is the starting state, A is an alphabet and $\delta : A \times \Sigma \times \mathbb{N} \to \Sigma \times \mathbb{N}$ is a deterministic transition function such that if $\delta(a, S, n) = (S', m)$, then $m \in \{n, n+1, n-1\} \cap \mathbb{N}$.

Recall that an event q is satisfiable if there is an initial segment $\mathfrak{N}_n$ of $\mathfrak{N}$ such that $\mathfrak{N}, n \models q$. The *event history of* q, $H(q)$, is the set of models for q in the above sense. For any q using at one instance of the LIFO/FIFO operator, we show how to construct a docm M_q such that $L(M) = H(q)$ (where $L(M)$ is the set of input histories accepted by the machine and $H(q)$ is the set of event histories for which q is true).

We show how to build docm for recognizing events in our algebra. We wish the docm to enter an accepting state every time the query q is true in the history.

Let us fix a language L of primitive events consisting of those events which occur in q_1 or q_2. The alphabet will consist of subsets of L (this is needed to model histories when events can occur simultaneously; otherwise single events as letters would do). We will construct the machines equivalent to q_1 and q_2 by recursion.

e where e is a primitive event. We define $M = (\Sigma, Acc, St, L, \delta)$ as follows. The machine has 3 states, a start state $S_1 = St$, an intermediate state S_2 and an accepting state $S_3 \in Acc$. The transition function δ is defined regardless of the counter which will remain empty during the transitions (and will be omitted below):

$$\delta(E, S_1) = S_3, \delta(F, S_1) = S_2, \delta(F, S_2) = S_2,$$
$$\delta(E, S_2) = S_3, \delta(E, S_3) = S_3, \delta(F, S_3) = S_2$$

where F and E are any subsets of L such that $e \in E$ and $e \notin F$.

$e_1 \parallel e_2$. Recursively build a docm $M_1 = (\Sigma_1, Acc_1, St_1, L, \delta_1)$ for recognizing e_1 and a docm $M_2 = (\Sigma_2, Acc_2, St_2, L, \delta_2)$ for recognizing e_2. We now build a docm $M = (\Sigma, Acc, St, L, \delta)$ which is the "product" of M_1 and M_2. Each state in M corresponds to a pair of states from M_1 and M_2: $\Sigma = \Sigma_1 \times \Sigma_2$. The start state of M corresponds to the start states of M_1 and M_2: $St = (St_1, St_2)$. The pairing of states and transitions then obey the following conditions $\delta(E, (S_i, T_j)) = (S_m, T_n)$ where $\delta_1(E, S_i) = S_m$ and $\delta_2(E, T_j) = T_n$. (S_i, T_j) is accepting for M iff S_i is accepting for M_1 and T_j is accepting for M_2: $Acc = Acc_1 \times Acc_2$.

$\sim e_1$. Build a docm for recognizing e_1. Change all accepting states into non-accepting states and vice versa (except for the start state which remains the same).

$e_1 \sqcup e_2$. Similar to $e_1 \parallel e_2$, but now (S_i, T_j) is accepting for M iff S_i is accepting for M_1 or T_j is accepting for M_2.

$e_1 \,;\, e_2$. Build a docm M_1 for recognizing e_1. Construct a new accept state for M_1 called S_a and $\delta(E, S_a) = S_a$ (for any set E of primitive events). For each accepting state $S_i \neq S_a$ in M_1, make it non accepting and alter the transitions such that $\delta(E, S_i) = S_a$ for any set E of primitive events. Call the new machine M_1'. Construct a docm M_2 for recognizing M_2. Now apply the "product" construction (used above for recognizing $e_1 \parallel e_2$) on M_1' and M_2. Intuitively, the state S_a has a "delaying" effect, the machine M will reach an accepting state if M_2 is in an accepting state (i.e., e_2 occurs now), and M_1 was in an accepting state before (i.e., e_1 occurred in the past).

$e_1 \,;_L\, e_2$. Build docm $M_1 = (\Sigma_1, Acc_1, St_1, L, \delta_1)$ and $M_2 = (\Sigma_2, Acc_2, St_2, L, \delta_2)$ for recognizing e_1 and e_2, respectively. Let Σ_1^+ be a set of the same cardinality as Σ_1 such that the two sets are disjoint. We denote the bijection from Σ to Σ^+ by $^+$. This is the only stage of the construction where we need the counter c. The definition of the new machine $M = (\Sigma, Acc, St, L, \delta)$ is a modification of the "product" construction. We define $\Sigma = (\Sigma_1 \cup \Sigma_1^+) \times \Sigma_2$, $St = (St_1, St_2)$ and $Acc = \{(S^+, T) : T \in Acc_2\}$. Initially, we let the counter c be 0. The definition of the transition function δ is as follows. In all cases below we assume that $\delta_1(S_i) = S_m$ and $\delta_2(T_j) = T_n$. We have

$$\delta(E, (S_i, T_j), c) = \begin{cases} ((S_m^+, T_n), c+1) & \text{if } S_m \in Acc_1 \text{ and } T_n \notin Acc_2 \\ ((S_m, T_n), c+1) & \text{if } S_m \in Acc_1,\ T_n \in Acc_2 \text{ and } c = 0 \\ ((S_m^+, T_n), c) & \text{if } S_m \in Acc_1,\ T_n \in Acc_2 \text{ and } c > 1 \\ ((S_m, T_n), c) & \text{if } S_m \notin Acc_1,\ T_n \in Acc_2 \text{ and } c = 0 \\ ((S_m^+, T_n), c-1) & \text{if } S_m \notin Acc_1,\ T_n \in Acc_2 \text{ and } c > 0 \\ ((S_m, T_n), c) & \text{if } S_m \notin Acc_1,\ T_n \notin Acc_2 \text{ and } c = 0 \\ ((S_m^+, T_n), c) & \text{if } S_m \notin Acc_1,\ T_n \notin Acc_2 \text{ and } c > 0; \end{cases}$$

and

$$\delta(E, (S_i^+, T_j), c) = \begin{cases} ((S_m^+, T_n), c+1) & \text{if } S_m \in Acc_1 \text{ and } T_n \notin Acc_2 \\ ((S_m^+, T_n), c) & \text{if } S_m \in Acc_1,\ T_n \in Acc_2 \text{ and } c > 0 \\ ((S_m, T_n), c+1) & \text{if } S_m \in Acc_1,\ T_n \in Acc_2 \text{ and } c = 0 \\ ((S_m, T_n), c) & \text{if } S_m \notin Acc_1 \text{ and } c = 0 \\ ((S_m^+, T_n), c) & \text{if } S_m \notin Acc_1,\ T_n \notin Acc_2 \text{ and } c > 0 \\ ((S_m^+, T_n), c-1) & \text{if } S_m \notin Acc_1,\ T_n \in Acc_2 \text{ and } c > 0. \end{cases}$$

$e_1 \,;_{MR}\, e_2$ and $e_1 \,;_C\, e_2$. These are similar to $e_1 \,;\, e_2$. Also, the proof would be virtually identical if $;_F$ had been chosen instead of $;_L$.

Now it is routine to check that the machine M we built for the event q has the required property: an n-long string of sets of primitive events is accepted iff the corresponding initial segment $\mathfrak{N}_n$ under the obvious evaluation of primitive events satisfies q. Further, the size of M is double exponential in terms of the size of q. Thus the above decision procedure works in triple exponential time. □

Corollary 1. *Let q be an event query in the algebra $\mathcal{E}_2$, such that it uses one instance of either $;_F$ or $;_L$. Then satisfiability of q is decidable.*

We now examine the problem of implication for event queries of the above type. The problem now becomes undecidable. Intuitively, this is because the individual queries can use their $;_L$ or $;_F$ operation in a co-operative fashion to simulate a machine with two counters.

Theorem 4. *Let q_1 and q_2 be event queries in the algebra $\mathcal{E}_2$ where each uses one instance of either $;_F$ or $;_L$. Then the problem of whether $q_1 \Rightarrow q_2$ is undecidable.*

Proof. We present a proof for the LIFO consumption semantics, it is virtually identical for FIFO. Given a Minsky machine [8] (abbreviated MM and defined below), we define a set of primitive and complex events used by an event query q, which checks whether the event history is a faithful representation of the computation of the MM. q is satisfiable iff the MM terminates. It also has the property that it can be rewritten in the form $q = q_1 \parallel\!\sim q_2$ and is thus satisfiable iff $\neg(q_1 \Rightarrow q_2)$. Thus, we are able to define two queries q_1 and q_2 (each using one instance of LIFO), which depend on the MM specification, and $q_1 \Rightarrow q_2$ iff the Minsky machine terminates — a problem which is undecidable.

An MM [8] is a sequence of n instructions $S_0 : com_0; S_1 : com_1; \ldots ; S_n : com_n$ where each instruction com_i has one of the following forms

$$\begin{array}{l} S_i : c_1 = c_1 + 1; goto\ S_j \\ S_i : c_2 = c_2 + 1; goto\ S_j \\ S_i : if\ c_1 = 0\ goto\ S_j\ else\ c_1 = c_1 - 1; goto\ S_k \\ S_i : if\ c_2 = 0\ goto\ S_j\ else\ c_2 = c_2 - 1; goto\ S_k \end{array}$$

where c_1 and c_2 are both counters. Execution begins at instruction S_0, with c_1 and c_2 initialised to zero. Thereafter, the machine executes the given instruction, appropriately changes the value of the given counter and jumps to the directed instruction. The machine halts iff it reaches instruction S_n. So, the computation of the machine can be understood as a (possibly infinite sequence) $S_{i_1}, S_{i_2}, \ldots$. We now define a number of events to ensure that the event history is a faithful representation of this sequence (if it is finite).

- For each machine instruction S_i, let e_{S_i} be an event.
- $e_c^1 = e_{push}^1 ;_L e_{pop}^1$ — used for maintaining the value of the 1st counter. Intuitively, e_{push}^1 increases it by one and e_{pop}^1 decreases it by one.
- $e_c^2 = e_{push}^2 ;_L e_{pop}^2$ — used for maintaining the value of the second counter.
- $e_{bad1}, e_{bad2}, e_{BAD1}, e_{BAD2}$ — events that occur if the history somehow deviates from the MM computation. Each uses multiple definitions and so they should in fact be interpreted as the disjunction of their individual definitions.
- e_{any} — this event is true at any point in the history. It can be written as the disjunction of all defined primitive events.

– $e_{first} = e_{any} \parallel\!\sim (e_{any}; e_{any})$ — the first event in the history.

We also rely on the following fact to guarantee correctness of the counters.

The event $e^1_c = e^1_{push} ;_L e^1_{pop}$ occurs iff the number of unconsumed instances of e^1_{push} was > 0 when e^1_{pop} occurred. Similarly for $e^2_c = e^2_{push} ;_L e^2_{pop}$.

Each point of the history corresponds to the machine being in a certain state.

$$\begin{aligned} e_{bad1} &= e_{any} \parallel\!\sim (e_{S_1} \sqcup e_{S_2} \sqcup \ldots \sqcup e_{S_n}) \\ e_{bad1} &= e_{S_i} \parallel e_{S_j} \quad (\forall i, j \in \{1, \ldots, n\}, i \neq j) \end{aligned}$$

Let N be the set of all instruction numbers, I_1 (I_2) be the set of all instruction numbers where counter one (two) is increased. Then for each $i \in I_1, j \in (N \setminus I_1)$, $a \in I_2$, $b \in (N \setminus I_2)$, we define

$$\begin{aligned} e_{bad1} &= (e_{S_i} \parallel\!\sim e^1_{push}) \sqcup (e_{S_j} \parallel e^1_{push}) \\ e_{bad2} &= (e_{S_a} \parallel\!\sim e^2_{push}) \sqcup (e_{S_b} \parallel e^2_{push}). \end{aligned}$$

Let D_1 be the set of all instruction numbers where counter one is tested by an *if*-statement and D_2 be the set of all instruction numbers where counter 2 is tested by an *if*-statement. Each such test is simulated by an e^1_{pop} or e^2_{pop} event occurring. So, for each $i \in D_1, j \in (N \setminus D_1)$,$a \in D_2$, $b \in (N \setminus D_2)$ we define

$$\begin{aligned} e_{bad1} &= (e_{S_i} \parallel\!\sim e^1_{pop}) \sqcup (e_{S_j} \parallel e^1_{pop}) \\ e_{bad2} &= (e_{S_a} \parallel\!\sim e^2_{pop}) \sqcup (e_{S_b} \parallel e^2_{pop}). \end{aligned}$$

We need to check that the correct 'gotos' of the machine are followed. Firstly:

$$e_{bad1} = e_{first} \parallel\!\sim e_{S_0}.$$

We will use the abbreviation $succ(e_a)$ as shorthand for the next event immediately following the occurrence of e_a: $succ(e_a) \equiv e_a ;_{MR} e_{any}$. For an instruction of the form $S_i : c_i = c_i + 1; goto\ S_j$ we define

$$e_{bad1} = succ(e_{S_i}) \parallel\!\sim e_{S_j}$$

For an instruction of the form $S_i : if\ c_1 = 0\ goto\ S_j\ else\ c_1 = c_1 - 1; goto\ S_k$

$$\begin{aligned} e_{bad1} &= succ(e_{S_i} \parallel\!\sim e^1_c) \parallel\!\sim e_{S_j} \\ e_{bad1} &= succ(e_{S_i} \parallel e^1_c) \parallel\!\sim e_{S_k}. \end{aligned}$$

Lastly, we ensure that if an e_{bad1} or e_{bad2} occurs at some point in the history, then it will also occur if and when the halt instruction is reached.

$$\begin{aligned} e_{BAD1} &= e_{bad1} \,;\, e_{S_n} \\ e_{BAD2} &= e_{bad2} \,;\, e_{S_n} \end{aligned}$$

We can now define the query q by

$$q = (e_{S_n} \parallel\!\sim e_{BAD1} \parallel\!\sim e_{BAD2}) = (q_1 \parallel\!\sim q_2)$$

where $q_1 = e_{S_n} \parallel\!\sim e_{BAD1}$ and $q_2 = e_{BAD2}$. Observe that q_1 doesn't depend on the event e^2_c and q_2 doesn't depend on the event e^1_c. Thus, they each use exactly one instance of the LIFO operator. □

Corollary 2. *Let q_1 be an event query in the event algebra $\mathcal{E}_2$ using two instances of either $;_F$ or $;_L$. Let q_2 be an event query written in $\mathcal{E}_1$. Then satisfiability of q_1 is undecidable and the problem of whether $q_1 \Leftrightarrow q_2$ is undecidable.*

5 Summary and Future Work

We have defined a simple core event language and defined a formal mechanism for specifying event consumption policies. We identified a class of policies for which equivalence and implication is in PSPACE. For more elaborate policies, we presented a class of event queries for which equivalence was decidable, provided each query used the policy only once. We then showed how this then became undecidable when testing for implication. In our future work, we plan to investigate a) the use of our results for understanding theories of natural numbers in fragments of first-order logic with order (e.g., the guarded fragment); b) the use of temporal logics such as CTL for expressing properties of interest in event languages and for active databases generally.

References

1. S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim. Composite events for active databases: semantics, contexts and detection. In *20th International Conference on Very Large Data Bases*, pages 606–617, 1994.
2. J. Chomicki and D. Toman. Implementing temporal integrity constraints using an active dbms. *IEEE TKDE*, 7(4):566–581, 1995.
3. O. Diaz, N.W. Paton, and J. Iturrioz. Formalizing and validating behavioral models through the event calculus. *Information Systems*, 23(3–4):179–196, 1998.
4. S. Gatziu and K. Dittrich. Detecting composite events in active database systems using Petri nets. In *Proceedings of the 4th International Workshop on Research Issues in Data Engineering*, pages 2–9, Houston, Texas, 1994.
5. N. Gehani, H. V. Jagadish, and O. Shmueli. Composite event specification in active databases: Model and implementation. In *VLDB'92*, pages 327–338, 1992.
6. A. Geppert and K. Dittrich. Performance assessment. In N. Paton, editor, *Active Rules in Database Systems*, pages 103–123. Springer–Verlag, 1999.
7. I. Hodkinson. On Gabbay's temporal fixed point operator. *Journal of Theoretical Computer Science*, 139:1–25, 1995.
8. M. Minsky. *Computation: Finite and Infinite Machines*. Prentice Hall, 1967.
9. I. Mirbel, B. Pernici, T. Sellis, S. Tserkezoglou, and M. Vazirgiannis. Checking temporal integrity of interactive multimedia documents. *The VLDB Journal*, 9(2):111–130, 2000.
10. I. Motakis and C. Zaniolo. Formal semantics for composite temporal events in active database rules. *Journal of Systems Integration*, 7(3–4):291–325, 1997.
11. N. Paton, editor. *Active Rules in Database Systems*. Springer-Verlag, 1999.
12. A. Sistla and E. Clark. Complexity of propositional linear temporal logics. *Journal of the ACM*, 32:733–749, 1985.
13. A. P. Sistla and O. Wolfson. Temporal triggers in active databases. *IEEE Transaction on KNowledge and Data Engineering*, 7(3):471–486, 1995.
14. L. G. Valiant and Paterson M. S. Deterministic one-counter automata. *Journal of Computer and System Sciences*, 10(3):340–250, 1975.
15. D. Zimmer and R. Unland. On the semantics of complex events in active database management systems. In *ICDE'99*, pages 392–399, 1999.