Specifying Event Logics for Active Databases

Iluju Kiringa
Department of Computer Science
University of Toronto, Toronto, Canada
kiringai@cs.toronto.edu

Abstract

Active databases are usually centered around the notion of Event-Condition-Action (ECA) rules. An ECA rule's action is executed upon detection of an event whenever the rule's condition is true. Events are traditionally described using an event algebra. Virtually every proposed active database management system (ADBMS) brings about a different event algebra. This makes it very difficult to analyze these proposals in a uniform way by spelling out what they may have in common, or how they may differ. Typically, logic might act as a framework for dealing with these issues. This paper treats events as (somewhat constrained) formulas of the situation calculus, a (second order) logic for reasoning about actions in general, and database updates and transactions in particular. We provide a framework for devising the semantics of complex events in this logic. Such semantics, formulated as theories of a special kind, are used for reasoning about the occurrence and consumption modes, which are an important dimension of events.

1 Introduction

An ADBMS captures the (re)active behavior of application domains by offering the possibility of automatic actions in response to relevant happenings called *events*. A reactive behavior involves an association of events with actions that should be performed automatically by the system once these events occur; it also involves a way of detecting the occurrences of events; it finally involves a specification of how the system should perform the actions associated with the events that may have occurred.

To be an ADBMS, a DBMS *should* necessarily support mechanisms for defining and managing ECA-rules by providing syntactic means for defining events, conditions, and actions; and it should also have a well-defined execution model capable of detecting event occurrences, evaluating conditions, and executing actions, having a well-defined execution semantics, and incorporating some user defined or predefined mechanism for resolving conflicts among rules whose events have occurred (triggered rules). An ADBMS is a DBMS extended with at least the mandatory dimensions of active behavior ([10]). So an ADBMS has two main components: a **reactive model**, also considered as a knowledge model ([10]), for defining events and associating them with actions; and an **execution model** for monitoring events and reacting to detected events.

The knowledge model is expressed in ECA-rules. These have three parts: event, condition, and action. Using an ADBMS rule language, one syntactically specifies the desired knowledge model of a database. Each of the parts of a rule is specified using a specific language: an event language for events, a condition language for conditions, and an action language for actions.

Events are traditionally described using an event algebra ([16]). Virtually every ADBMS brings about a different event algebra. This makes it very difficult to analyze these proposals in a uniform way by spelling out what they may have in common, or how they may differ. Logic might act as a framework for dealing with these issues. However, which logic is suited for such a task? Since database events occur in a domain whose dynamics is the result of the execution of actions, it seems natural to pick a logic for reasoning about actions for the task of capturing event algebras. In this paper, we argue and show that the situation calculus ([9],[12]) can act as such a logic. With such a logic in hand, one may tackle the issue of comparing two given event algebras by first capturing them using theories of actions of a suitable form. If one has a particularly desirable property that both algebras have to fulfill, then one has to formulate it as a sentence of the situation calculus. Now, one may check, for example, whether this sentence is entailed by the theories of actions capturing the two algebras. In this way, one knows that both algebras could be chosen for one's purposes. To give an example of the usefulness of our approach, we will, among other things, find out how difficult it can be to establish whether two or more events are the same.

We treat complex events as (somewhat constrained) formulas of the situation calculus, a (second order) logic for reasoning about actions in general, and database updates and transactions in particular. We provide a framework for devising the semantics of complex events in this logic. Such semantics, formulated as theories of a special kind, are used for reasoning about the occurrence and consumption modes, an important dimension of events. In this setting, we define event detection, and find out that this task is formally identical to posing a situation calculus query against a background theory representing the active database domain. As such, event detection amounts to establishing a logical entailment of the posed query from the background theory.

2 Situation Calculus

The situation calculus is a many-sorted second order language with equality specifically designed for representing dynamically changing worlds. We consider three sorts for the following:

Actions: These are first order terms consisting of a $n-ary$ action function symbol. In modeling ADBMS, they typically will correspond to database updates or transactional actions.[1]

Situations: These are first order terms denoting a sequence of actions. These sequences are represented using a binary function symbol do; $do(\alpha, s)$ denotes the sequence resulting from adding the action α to the sequence s. The special constant S_0 denotes the *initial situation*, namely the empty action sequence [ ]. In modeling ADBMS, situations will correspond to database logs which are sequences of primitive database actions.

Objects: These constitute a catch-all sort representing everything else depending on the database domain of application.

The language also includes a finite number of predicates called *fluents*, which represent the database relations and whose truth values vary from situation to situation. Fluents are denoted by predicate symbols with last argument a situation term. For any fluent $F(\vec{x}, s)$, we shall have database update actions $F\_insert(\vec{x}, t)$ and $F\_delete(\vec{x}, t)$ with the obvious meaning; the argument t denotes the transaction that issues the update action.

Finally, the language includes special predicates $Poss$, and $\sqsubset$; $Poss(a, s)$ means that the action a is possible in the situation s, and $s \sqsubset s'$ states that the situation s' is reachable from s by performing some sequence of actions. In ADBMS terms, $s \sqsubset s'$ means that s is a proper sublog of the log s'. The logical symbols of the language are $\neg$ and $\exists$. Other logical symbols are introduced as abbreviations in the usual way.

[1] Transactional actions are, e.g., $Begin$, $Commit$, $Rollback$, etc.

A database domain is axiomatized in the situation calculus with axioms which describe *how* and under what *conditions* the domain is changing or not changing as a result of performing actions. Such axioms are called basic action theory in [12] and have been extended to *relational theories* in [7].[2] The later comprise the following: domain independent foundational axioms for situations; integrity constraint axioms; action precondition axioms, one for each database update, and one for each transactional action, stating the conditions of change; successor state axioms, one for each fluent, stating how change occurs; dependency axioms, stating how database transactions interact with each other; unique names axioms for action terms; and axioms describing the initial situation. Finally, by convention, a free variable will always be implicitly bound by a prenex universal quantifier.

Example 1 *Consider a stock trading database ([14]), whose schema has the following relations:* $price(stock\_id, price, time, trans, s)$, $stock(stock\_id, price, closingprice, trans, s)$, *and* $customer(cust\_id, balance, stock\_id, trans, s)$*, which are relational fluents. The explanation of the attributes is as follows:* $stock\_id$ *is the identification number of a stock,* $price$ *the current price of a stock,* $time$ *the pricing time,* $closingprice$ *the closing price of the previous day,* $cust\_id$ *the identification number of a customer,* $balance$ *the balance of a customer, and* $trans$ *is a transaction identifier.* $\square$

The database domain described in Example 1 can be axiomatized as the following relational theory.[3] We may enforce any set of tractable functional dependencies such as primary key constraints; e.g.,

$$stock(st\_id, price, closPr, s) \wedge stock(st\_id, price', closPr', s) \supset pr = pr', closPr = closPr';$$

and we can verify the integrity constraint

$$customer(cust\_id, bal, stock\_id, s) \supset bal \geq 0$$

at the end of transactions.

Unique name axioms are given as in basic action theories. A sample initial database axiom is:

$$\begin{aligned} price(s\_id, price, time, trans, S_0) \equiv\; & s\_id = ST_1 \wedge pr = \$100 \wedge t = 100100 : 4PM \vee \\ & s\_id = ST_2 \wedge pr = \$110 \wedge t = 100100 : 9AM \vee \\ & s\_id = ST_3 \wedge pr = \$50 \wedge t = 100100 : 1PM. \end{aligned}$$

The following is the action precondition axiom for the update $price\_insert$:

$$\begin{aligned} Poss(price\_insert(stock\_id, price, time, t), s) \equiv\; & \neg(\exists t')price(stock\_id, price, time, t', s) \wedge \\ & IC(do(price\_insert(stock\_id, price, time, t), s)) \wedge running(t, s). \end{aligned}$$

This says that the database update $price\_insert$ is possible in situation s iff the tuple to be inserted is not already in the appropriate relation, the integrity constraints IC are enforced in the resulting situation, and of course the transaction executing the update is running. Similar

[2] In fact, relational theories were first used in [11], where, however, they did not capture the dynamics of database domains.

[3] We omit precondition axioms for transactional actions and the dependency axioms; they play no role in the sequel of this paper.

axioms are given for $stock\_insert$, $stock\_delete$, $customer\_insert$, and $customer\_delete$. The successor state axiom for the fluent $price$ is:

$$
\begin{aligned}
&price(stock\_id, price, time, t, do(a, s)) \equiv \\
&\quad \{[(\exists t_1)(a = price\_insert(stock\_id, price, t_1, time) \vee \\
&\quad\ (\exists t_2) price(stock\_id, price, time, t_2, s) \wedge \\
&\quad\ \neg(\exists t_3) a = price\_delete(stock\_id, price, t_3, time)) \wedge \neg(\exists t')a = Rollback(t')] \vee \\
&\quad\ (\exists t')[a = Rollback(t') \wedge RestoreBegin(price, (stock\_id, price, time), t', s)]\}.
\end{aligned}
$$

This intuitively means that the tuple $(stock\_id, price, time)$ is in the relation $price$ in the situation $do(a, s)$, which results from the execution of the action a in the situation s, when the action a is $price\_insert$ or the tuple was already there in the situation s and was not deleted (a is not $price\_delete$); but when a is a rollback action, then the relation $price$ is reset to the value it had at the beginning of the transaction that modified it. Reseting values of fluents in that way is captured by $RestoreBegin(F, \vec{x}, t, s)$ which means that F gets the value it had at the beginning of transaction t for the arguments $\vec{x}$. Successor state axioms for fluents $stock$ and $customer$ are similar.

Let $\mathcal{D}$ be a relational theory for some domain, as described above, and let $Q(s)$ be a situation calculus formula – the *query* – with one free situation variable s. Moreover, let the log $S = do(\alpha_n, do(\alpha_{n-1}, \cdots, do(\alpha_1, S_0) \cdots))$ be a ground situation term, that is, one that mentions no free variables. We define the querying problem in the situation calculus as the problem of determining whether $\mathcal{D} \models Q(S)$. The *answer* to Q relative to S is "yes" iff $\mathcal{D} \models Q(S)$. The answer is "no" iff $\mathcal{D} \models \neg Q(S)$. Let $\mathcal{D}$ denote the relational theory of Example 1. Then the following is a sample query:

$$
\begin{aligned}
\mathcal{D} \models customer(Ray, \$100000, ST1, T, do(customer\_insert(Ray, \$100000, ST1, T),& \\
do(price\_insert(ST_1, \$100, 100100 : 4PM)))),&
\end{aligned}
$$

which is the question whether the theory $\mathcal{D}$ logically entails the fact that customer Ray has a balance of $\$100000$ and stocks of $ST1$ in the situation following the execution of the updates $price\_insert(ST_1, \$100, 100100 : 4PM)$ and $customer\_insert(Ray, \$100000, ST1, T)$ in this order.

In [8], GOLOG, a situation calculus-based programming language, is introduced for defining complex actions in terms of a set of primitive actions axiomatized as basic action theories. In [6, 7], relational theories are used as background axioms. Among other constructs — mainly the standard control structures of most Algol-like languages —, GOLOG has test actions of the form $\phi?$, i.e., test the truth value of expression ϕ in the current situation. Thus, given an actual situation s, a GOLOG interpreter will in fact pose a situation calculus query $\phi(s)$ to test the truth value of ϕ in s. At the GOLOG language level, ϕ is a *situation suppressed formula*, that is, one in which the situation arguments of all fluents have been removed.

3 Specifying the Reactive Model

3.1 ECA-Rules

An ECA rule is a construct of the following form:

$$< t : R : \tau : \zeta(\vec{x}) \rightarrow \alpha(\vec{y}) > . \tag{1}$$

In this construct, t specifies the transaction that fires the rule, τ specifies the event that triggers the rule, and R is a constant giving the rule's identification number (or name). A rule

$<trans : Update\_stocks : price\_inserted :$
$(\exists c, time, bal, price')[price\_inserted(s\_id, price, time) \wedge$
$customer(c, bal, s\_id) \wedge stock(s\_id, price', clos\_pr)]$
$\rightarrow$
$stock\_insert(s\_id, price, clos\_pr) >$

$<trans : Buy\_100shares : price\_inserted :$
$(\exists\ new\_price, time, bal, pr, clos\_pr)[price\_inserted(s\_id, new\_price, time) \wedge$
$customer(c, bal, s\_id) \wedge stock(s\_id, pr, clos\_pr) \wedge new\_price < 50 \wedge clos\_pr > 70]$
$\rightarrow$
$buy(c, s\_id, 100) >$

Figure 1: Rules for updating stocks and buying shares

is triggered if the event specified in its event part occurs. The relationship between the event occurrence and the triggering of a rule is dictated by *consumption modes*, which determine when an event that occurred will cease to be considered as having occurred. In its simplest form, the semantics of event consumption is that a rule is triggered if the event specified in its event part occurred since the beginning of the open transaction in which that event part is evaluated. Events are one of the predicates $F\_inserted(r, t, s)$ and $F\_deleted(r, t, s)$, called *event fluents*, or a combination thereof using logical connectives. The ζ part specifies the rule's condition; it mentions predicates $F\_inserted(r, \vec{x}, t, s)$ and $F\_deleted(r, \vec{x}, t, s)$ called *transition fluents*, which denote the transition tables ([14]) corresponding to insertions into and deletions from the relation F.[4] In (1), arguments t, R, and s are suppressed from all the fluents; the two first ones are restored when (1) is translated to a GOLOG program, and s is restored at run time. Finally, α gives a GOLOG program which will be executed upon the triggering of the rule once the specified condition holds. Actions also may mention transition fluents. Notice that $\vec{x}$ are free variables mentioned by ζ and contain all the free variables $\vec{y}$ mentioned by α.

Example 2 *Consider the following active behavior for Example 1. Each customer's stock is updated whenever new prices are notified. When current prices are being updated, the closing price is also updated if the current notification is the last of the day; moreover, suitable trade actions are initiated if some conditions become true of the stock prices, under the constraint that balances cannot drop below a certain amount of money. Two rules for this active behavior are shown in Figure 1.* $\square$

In what follows, it is appropriate to define what counts as a term or a formula whose rule and transaction arguments have been either suppressed or restored. we introduce the concepts of *rule id and transaction id suppressed* terms and formulas, and *rule id and transaction id restored* terms and formulas, respectively.

Definition 1 *The rid and tid suppressed-terms (rts-terms) and formulas (rts-formulas) of* $\mathfrak{R}$ *are terms / formulas in which the rid and tid arguments of fluents have been removed.*

[4] Transition tables stores the tuples inserted into or deleted from relations.

Definition 2 *The rid and tid restored-terms (rtr-terms) and formulas (rtr-formulas) are terms / formulas in which previously removed rid and tid arguments of fluents have been restored. Whenever t and ϕ are rts-term and rts-formula, respectively, and rid and tid are rule and transaction identifiers, respectively, we use the notation $t[rid, tid]$ and $\phi[rid, tid]$ to denote the corresponding rtr-term and rtr-formula, respectively.*

With reference to the syntax of an ECA rule (see (1)), the notation $\tau(\vec{x})[r, t]$ means the result of restoring the arguments r and t to all event fluents mentioned by τ, and $\zeta(\vec{x})[r, t]$ means the result of restoring the arguments r and t to all transition fluents mentioned by ζ. For example, if τ is the complex event

$$price\_inserted \wedge customer\_inserted,$$

then $\tau[r, t]$ is

$$price\_inserted(r, t) \wedge customer\_inserted(r, t).$$

3.2 Transition Fluents and Neteffect Policy

To characterize the notions of transition tables and events, we introduce the fluent $considered(r, t, s)$ which intuitively means that the rule r can be considered for execution in situation s with respect to the transaction t. The following gives an abbreviation for $considered(r, t, s)$:

$$considered(r, t, s) =_{df} (\exists t').running(t', s) \wedge ancestor(t', t, s). \tag{2}$$

Intuitively, this means that, as long as an ancestor of t is running, any rule r may be considered for execution. In actual systems this concept is more sophisticated than this scheme.[5]

For each database fluent $F(\vec{x}, t, s)$, we introduce the *transition fluents* $F\_inserted(r, \vec{x}, t, s)$ and $F\_deleted(r, \vec{x}, t, s)$. The following successor state axioms characterizes them:

$$\begin{aligned} &F\_inserted(r, \vec{x}, t, do(a, s)) \equiv considered(r, t, s) \wedge (\exists t')a = F\_insert(\vec{x}, t')] \vee \\ &\quad F\_inserted(r, \vec{x}, t, s) \wedge \neg a = F\_delete(\vec{x}, t). \end{aligned} \tag{3}$$

$$\begin{aligned} &F\_deleted(r, \vec{x}, t, do(a, s)) \equiv considered(r, t, s) \wedge (\exists t')a = F\_delete(\vec{x}, t')] \vee \\ &\quad F\_deleted(r, \vec{x}, t, s) \wedge \neg a = F\_insert(\vec{x}, t). \end{aligned} \tag{4}$$

Axiom (3) means that a tuple $\vec{x}$ is considered inserted in situation $do(a, s)$ iff the internal action $F\_insert(\vec{x}, t')$ was executed in the situation s while the rule r was considered, or it was already inserted and a is not the internal action $F\_delete(\vec{x}, t')$; here, t' is a transaction that can be different than t. This captures the notion of *net effects* ([14]) of a sequence of actions. The net effects are policies for accumulating only changes that really affect the database, thus ignoring transient changes; in this case, if a record is deleted after being inserted, this amounts to nothing having happened. Further net effect policies can be captured in this axiom, but we leave this topic out of this paper.

3.3 Primitive and Complex Event Fluents

Events that trigger ECA rules are generally associated with the data manipulation language of the underlying database. In the situation calculus, for each database fluent $F(\vec{x}, t, s)$, we introduce the *primitive event fluents* $F\_inserted(r, t, s)$ and $F\_deleted(r, t, s)$.

[5] For example, in Starburst ([14]), r will be considered in the future course of actions only from the time point where it last stopped being considered.

The primitive event fluent $F\_inserted(r, t, s)$ corresponding to an insertion into the relation F has the following successor state axiom:

$$F\_inserted(r, t, do(a, s)) \equiv (\exists t')a = F\_insert(\vec{x}, t') \wedge considered(r, t, s) \vee \\ F\_inserted(r, t, s). \tag{5}$$

The primitive event fluent $F\_deleted(r, t, s)$ corresponding to a deletion from the relation F has a similar successor state axiom:

$$F\_deleted(r, t, do(a, s)) \equiv (\exists t')a = F\_delete(\vec{x}, t') \wedge considered(r, t, s) \vee \\ F\_deleted(r, t, s). \tag{6}$$

Definition 3 (Primitive Event Occurrence) *A primitive event e occurs in situation s with respect to a rule r and a transaction t iff $\mathcal{D} \models e[r, t, s]$. Here $\mathcal{D}$ is a relational theory incorporating the successor state axioms for the primitive event fluents.*

So, on this definition, an event occurrence (or, equivalently, event detection) is a situation calculus query in the sense of Section 2. Following [2], we call this an *event query*.

As stated earlier, complex events are usually built from simpler, and ultimately, the primitive ones using some event algebra ([2], [16]). Using logical means, we now specify the semantics of complex events that accounts for the active dimension of consumption mode. This development will ultimately lead to a logic for events, instead of an algebra.

That complex events are built from simpler ones is just one of the intuitive assumptions that one can make about events. In [16], Zimmer and Unland make five basic assumptions about events, which we adopt in the context of the situation calculus as follows:

– Events are interpreted over a set of situations (logs).
– Primitive events are detected at situations, in the order at which they occurred.
– Complex events are built from primitive ones (components) using logical connectives, and many complex events can independently be built from the same set of simpler ones.
– The situation at which a complex event occurs is that situation at which the very last ("last" in the sense of the ordering of situation mentioned above) of its components occurs.
– Many events may occur at the same situation, that is, simultaneously.

In order to build complex events, we use the usual logical connectives and symbols $\wedge$, $\vee$, $\neg$, $\forall$, as well as the ordering predicate $\sqsubset$. These logical symbols and predicates will be used to introduce complex events in the form of abbreviations. The following fluents express some basic constructs for building complex events: $seq\_ev(r, t, e_1, e_2, s)$, $simult\_ev(r, t, e_1, e_2, s)$, $conj\_ev(r, t, e_1, e_2, s)$, $disj\_ev(r, t, e_1, e_2, s)$, and $neg\_ev(r, t, e, s)$. Table 1 gives the informal semantics of these fluents.

In the absence of consumption modes, the formal situation calculus based-semantics of complex events in terms of simpler ones is as follows:

$$neg\_ev(r, t, e, s) =_{df} (\exists r')\neg e[r', t, s], \tag{7}$$

$$seq\_ev(r, t, e_1, e_2, s) =_{df} (\exists r')e_2[r', t, s] \wedge (\exists r'', s').s' \sqsubset s \wedge e_1[r'', t, s'], \tag{8}$$

$$simult\_ev(r, t, e_1, e_2, s) =_{df} (\exists r')e_1[r', t, s] \wedge (\exists r'')e_2[r'', t, s], \tag{9}$$

$$conj\_ev(r, t, e_1, e_2, s) =_{df} (\exists r_1)seq\_ev(r_1, t, e_1, e_2, s) \vee \\ (\exists r_2)seq\_ev(r_2, t, e_2, e_1, s) \vee (\exists r_3)simult\_ev(r_3, t, e_1, e_2, s), \tag{10}$$

$$disj\_ev(r, t, e_1, e_2, s) =_{df} (\exists r')e_1[r', t, s] \vee (\exists r'')e_2[r'', t, s]. \tag{11}$$

Definition 4 (Complex Event Occurrence) *A complex event e occurs in situation s with respect to a rule r and a transaction t iff $\mathcal{D} \models e[r, t, s]$. Here $\mathcal{D}$ is a relational theory incorporating the abbreviations above for the complex event fluents.*

Fluent	Informal semantics
$seq\_ev(r,t,e_1,e_2,s)$	event e_1 occurs before event e_2 in s
$simult\_ev(r,t,e_1,e_2,s)$	events e_1 and e_2 occur simultaneously in s
$conj\_ev(r,t,e_1,e_2,s)$	events e_1 and e_2 occur together in any order in s
$disj\_ev(r,t,e_1,e_2,s)$	either event e_1 or event e_2 occurs in s
$neg\_ev(r,t,e,s)$	event e does not occur in s

Table 1: Informal semantics of basic complex events

Following [16], we emphasize the following good language design principles with respect to complex events of any logic for events: **minimality**, i.e., the logic must provide a very small minimal core of unambiguous constructs; **symmetry**, i.e., the constructs must semantically be context free; **orthogonality**, i.e., the core language must be expressive enough.

From the basic constructs (7)–(11) above, the set $\{seq\_ev(r,t,e_1,e_2,s), e_1, \cdots, e_n\}$ is the minimal core from which all the others complex events are built, where the e_i, $i = 1, \cdots, n$, are primitive event fluents. Any other construct not belonging to that core must satisfy the good language design principles of symmetry and orthogonality listed above.

3.4 Event Fluents and Consumption Modes

Once we have specified a way of building a complex event e from simpler ones, we still have to specify which occurrences of the component of e must be selected in order for e to occur (*event occurrence selection*), and what to do with those occurrences once they have been used in the occurrence of e (*occurrence consumption*). *Consumption modes* are used to determine the event occurrence selection and consumption of the events.

Presumably, it suffice to assign consumption modes to the minimal core $\{seq\_ev(r,t,e_1,e_2,s), e_1, \cdots, e_n\}$ of the logic for events.

As for primitive event fluents, occurrence selection is trivial: from axioms (5) and (6) we see clearly that the first occurrence of a primitive event fluent may trigger any considered ECA rule. From axioms (5) and (6), we also see that a primitive event fluent remains unconsumed for any later considered rule. So this way we achieve a no-consumption scope. To achieve a global consumption scope, we must change (5) and (6) respectively to

$$F\_inserted(r,t,do(a,s)) \equiv (\exists t')a = F\_insert(\vec{x},t') \wedge considered(r,t,s), \qquad (12)$$

and

$$F\_deleted(r,t,do(a,s)) \equiv (\exists t')a = F\_delete(\vec{x},t') \wedge considered(r,t,s). \qquad (13)$$

A particular consumption mode is imposed upon the sequence fluent $seq\_ev(r,t,e_1,e_2,s)$ by defining a conjunct $\Psi_{CM}(t,\vec{s},s)$ such that

$$seq\_ev(r,t,e_1,e_2,s) =_{df} (\exists \vec{s})\Psi_{seq}(t,\vec{s},s) \wedge \Psi_{CM}(t,\vec{s},s), \qquad (14)$$

where $\Psi_{seq}(t,\vec{s},s)$ is a situation calculus formula specifying the semantics of $seq\_ev(r,t,e_1,e_2,s)$ (i.e, the right-hand side of (8)); $\Psi_{CM}(t,\vec{s},s)$ is a situation calculus formula that specifies the consumption mode used.

If $\mathcal{L}$ is a distinguished fragment of the situation calculus such that $\Psi_{CM}(t, \vec{s}, s) \in \mathcal{L}$, then this induces the *consumption mode class* $CM_{\mathcal{L}}$. In general, $\mathcal{L}$ can be any fragment of the situation calculus. Of course, formulas $\Psi_{CM}(t, \vec{s}, s)$ used in practice must belong to logics $\mathcal{L}$ that enjoy particularly desirable properties (e.g., tractability) with respect to specific problems such as the equivalence of two given complex events ([2]).

To deal with consumption modes for sequences, we introduce further terminology adapted from [16]. Suppose $e = seq\_ev(r, t, e_1, e_2, s)$; then e_1 is called the *initiator* and e_2 the *terminator* of e. An component e' of a sequence e is said to be *consumed* iff it no longer can contribute to the detection of e.

By virtue of the Zimmer-Unland assumptions about events, a sequence $seq\_ev(r, t, e_1, e_2, s)$ occurs when its terminator e_2 occurs, provided that its initiator occurred according to a given consumption mode.

Some possible consumption modes for event sequences are[6]:

First: Selects the oldest occurrence of the initiator, after which this occurrence is consumed.
Consumed Last: Selects the most recent occurrence of the initiator, after which this occurrence is consumed.
Non-Consumed Last: Selects the most recent occurrence of the initiator, which remains unconsumed as long as there is no occurrence of the initiator.
Cumulative: Selects all occurrences of the initiator up to the situation where the terminator occurs, after which all these occurrences of the initiator are consumed.
FIFO: Selects the earliest occurrence of the initiator that has not yet been consumed, after which this occurrence is consumed.
LIFO: Selects the latest occurrence of the initiator that has not yet been consumed, after which this occurrence is consumed.

Now we spell out details of these consumption modes.

First. Take $\Psi_{CM}(t, \vec{s}, s)$ in (14) as

$$(\forall s^*).s' \sqsubset s^* \sqsubset s \supset ((\exists r_1)e_1[r_1, t, s^*] \vee \neg(\exists r_2)e_2[r_2, t, s^*]). \tag{15}$$

So to detect the sequence under this mode, we have to establish the entailment

$$\begin{aligned}\mathcal{D} \models &(\exists r')e_2[r', t, s] \wedge (\exists s', r'')[s' \sqsubset s \wedge e_1[r'', t, s'] \wedge \\ &(\forall s^*).s' \sqsubset s^* \sqsubset s \supset ((\exists r_1)e_1[r_1, t, s^*] \vee \neg(\exists r_2)e_2[r_2, t, s^*])].\end{aligned} \tag{16}$$

Consumed Last. We express this by taking $\Psi_{CM}(t, \vec{s}, s)$ in (14) as

$$(\forall s^*).s' \sqsubset s^* \sqsubset s \supset \neg(\exists r_1)e_1[r_1, t, s^*] \wedge \neg(\exists r_2)e_2[r_2, t, s^*]. \tag{17}$$

Therefore, detecting the sequence under this mode amounts to establishing the entailment

$$\begin{aligned}\mathcal{D} \models &(\exists r')e_2[r', t, s] \wedge (\exists s', r'')[s' \sqsubset s \wedge e_1[r'', t, s'] \wedge \\ &(\forall s^*).s' \sqsubset s^* \sqsubset s \supset \neg(\exists r_1)e_1[r_1, t, s^*] \wedge \neg(\exists r_2)e_2[r_2, t, s^*]].\end{aligned} \tag{18}$$

Non-Consumed Last. We express this by taking $\Psi_{CM}(t, \vec{s}, s)$ in (14) as

$$(\forall s^*).s' \sqsubset s^* \sqsubset s \supset \neg(\exists r_1)e_1[r_1, t, s^*]. \tag{19}$$

So to detect the sequence under this mode, we have to establish the entailment

$$\begin{aligned}\mathcal{D} \models &(\exists r')e_2[r', t, s] \wedge (\exists s', r'')[s' \sqsubset s \wedge e_1[r'', t, s'] \wedge \\ &(\forall s^*).s' \sqsubset s^* \sqsubset s \supset \neg(\exists r_1)e_1[r_1, t, s^*]].\end{aligned} \tag{20}$$

[6] In [16] and [2] some of these are used, sometimes under different names.

Cumulative: Here, we take $\Psi_{CM}(t, \vec{s}, s)$ in (14) as

$$(\forall s^*).s' \sqsubset s^* \sqsubset s \supset \neg(\exists r_2)e_2[r_2, t, s^*]. \tag{21}$$

So to detect the sequence under this mode, we have to establish the entailment

$$\begin{aligned}\mathcal{D} \models &(\exists r')e_2[r', t, s] \wedge (\exists s', r'')[s' \sqsubset s \wedge e_1[r'', t, s'] \wedge \\ &(\forall s^*).s' \sqsubset s^* \sqsubset s \supset \neg(\exists r_2)e_2[r_2, t, s^*]].\end{aligned} \tag{22}$$

FIFO: Here, $\Psi_{CM}(t, \vec{s}, s)$ in (14) is

$$\begin{aligned}&(\forall s^*)[s^* \sqsubset s \supset \neg((\exists r_1)e_1[r_1, t, s^*] \wedge seq\_ev(r, t, e_1, e_2, s^*))] \wedge \\ &(\forall s^*)[s^* \sqsubset s' \supset [(\exists r_1)e_1[r_1, t, s^*] \supset (\exists s^{**})s^{**} \sqsubset s \wedge seq\_ev(r, t, e_1, e_2, s^{**})]].\end{aligned} \tag{23}$$

So to detect the sequence under this mode, we must establish the entailment

$$\begin{aligned}\mathcal{D} \models &(\exists r')e_2[r', t, s] \wedge (\exists s', r'')[s' \sqsubset s \wedge e_1[r'', t, s'] \wedge \\ &(\forall s^*)[s^* \sqsubset s \supset \neg(e_1[r, t, s^*] \wedge seq\_ev(r, t, e_1, e_2, s^*))] \wedge \\ &(\forall s^*)[s^* \sqsubset s' \supset [(\exists r_1)e_1[r_1, t, s^*] \supset (\exists s^{**})s^{**} \sqsubset s \wedge seq\_ev(r, t, e_1, e_2, s^{**})]]]].\end{aligned} \tag{24}$$

LIFO: Here, $\Psi_{CM}(t, \vec{s}, s)$ in (14) is

$$\begin{aligned}&(\forall s^*)[s^* \sqsubset s \supset \neg((\exists r_1)e_1[r_1, t, s^*] \wedge seq\_ev(r, t, e_1, e_2, s^*))] \wedge \\ &(\forall s^*)[s' \sqsubset s^* \sqsubset s \supset [(\exists r_1)e_1[r_1, t, s^*] \supset (\exists s^{**})s^{**} \sqsubset s \wedge seq\_ev(r, t, e_1, e_2, s^{**})]].\end{aligned} \tag{25}$$

So to detect the sequence under this mode, we must establish the entailment

$$\begin{aligned}\mathcal{D} \models &(\exists r')e_2[r', t, s] \wedge (\exists s', r'')[s' \sqsubset s \wedge e_1[r'', t, s'] \wedge \\ &(\forall s^*)[s^* \sqsubset s \supset \neg(e_1[r, t, s^*] \wedge seq\_ev(r, t, e_1, e_2, s^*))] \wedge \\ &(\forall s^*)[s' \sqsubset s^* \sqsubset s \supset [(\exists r_1)e_1[r_1, t, s^*] \supset (\exists s^{**})s^{**} \sqsubset s \wedge seq\_ev(r, t, e_1, e_2, s^{**})]]].\end{aligned} \tag{26}$$

For the purpose of characterizing (some of) the consumption modes, the set of operators of first order past temporal logic can be introduced using a set of appropriate abbreviations as follows ([1]):

Definition 5 (First order past temporal logic)

$$\begin{aligned}&previously(\phi, s) =_{df} (\exists s')(\exists a).s = do(a, s') \wedge \phi(s'), \\ &past(\phi, s) =_{df} (\exists s').S_0 \sqsubseteq s' \sqsubset s \wedge \phi(s'), \\ &always(\phi, s) =_{df} (\exists s').S_0 \sqsubseteq s' \sqsubset s \supset \phi(s'), \\ &since(\phi, \psi, s) =_{df} (\exists s')[S_0 \sqsubseteq s' \sqsubset s \wedge \psi(s') \wedge (\forall s'').s' \sqsubset s'' \sqsubseteq s \supset \phi(s')].\end{aligned}$$

First order past temporal formulas expressed in the situation calculus are formulas that may include the logical connectives $\neg$, $\wedge$, $\vee$ and $\supset$, quantification over individuals of sort $objects$, and the predicates abbreviated above. In the abbreviations above, ϕ and ψ are first order past temporal formulas.

In the context of the situation calculus, the whole development above leads to the concept of an *event logic* which we now formally express as a definition.

Definition 6 (Event Logic) *An event logic is a triple $(E, C, \mathcal{L})$, where E is a set of event fluents, C is a set of event connectives, together with the predicate $\sqsubset$, and $\mathcal{L}$ is a fragment of the situation calculus specifying the consumption mode associated with event sequences.*

Once we have an event logic and a set of events, we would want to establish whether these events are the same. This is to give an example of the usefulness of our approach. We would also want to find out how difficult it can be to establish whether two or more events are the same. The following definition and theorem serve this purposes.

Definition 7 (Implication and Equivalence Problems for an Event Logic) *Suppose $e[r, t, s]$ and $e'[r, t, s]$ are two events of a given event logic $\mathcal{E}$. Then the implication and equivalence problems for $\mathcal{E}$ are the problems of establishing whether, for given R and T, $\mathcal{D} \models (\forall s).e[R, T, s] \supset e'[R, T, s]$, and $\mathcal{D} \models (\forall s).e[R, T, s] \equiv e'[R, T, s]$, respectively. Here $\mathcal{D}$ specifies the semantics of events according to the event logic $\mathcal{E}$.*

Suppose that $seq\_ev^F(r, t, e_1, e_2, s)$, $seq\_ev^{CL}(r, t, e_1, e_2, s)$, $seq\_ev^{NL}(r, t, e_1, e_2, s)$, and $seq\_ev^{CUM}(r, t, e_1, e_2, s)$ denote event sequence fluents with the consumption modes First, Consumed-Last, Non-Consumed-Last, and Cumulative, whose semantics have been given above. Then we have the following result:[7]

Theorem 1 *Suppose $\mathcal{E} = (E, C, \mathcal{L})$ is the event logic given by:*

- $E = \{F\_inserted(r, t, s), F\_inserted(r, t, s), seq\_ev^{CM}(r, t, e_1, e_2, s),$
  $simult\_ev(r, t, e_1, e_2, s), conj\_ev(r, t, e_1, e_2, s), disj\_ev(r, t, e_1, e_2, s),$
  $neg\_ev(r, t, e, s)\}$,
  with $CM \in \{F, CL, NL, CUM\}$;
- $C = \{\neg, \wedge, \sqsubset\}$;
- *$\mathcal{L}$ is the past temporal fragment of the situation calculus.*

Then both the implication and the equivalence problems for $\mathcal{E}$ are in PSPACE.

4 Active Relational Theories

An *active relational language* is a relational language extended in the following way: for each $n{+}2$-ary fluent $F(\vec{x}, t, s)$, we introduce two $n{+}3$-ary transition fluents $F\_inserted(r, \vec{x}, t, s)$ and $F\_deleted(r, \vec{x}, t, s)$, and two 3-ary event fluents $F\_inserted(r, t, s)$ and $F\_deleted(r, t, s)$.

Definition 8 (Active Relational Theory) *A theory $\mathcal{D} \subseteq \mathfrak{W}$ is an* active relational theory *iff it is of the form $\mathcal{D} = \mathcal{D}_{brt} \cup \mathcal{D}_{tf} \cup \mathcal{D}_{ef}$, where*

1. *$\mathcal{D}_{brt}$ is a relational theory.*[8]
2. *$\mathcal{D}_{tf}$ is the set of axioms for transition fluents.*

[7] Let us briefly mention the proof ideas. Theorem 1 is a corollary of the fact that the complexity of propositional temporal linear logic with operators $Since$ and $Previously$ is in PSPACE (See [13] for similar results for propositional future temporal logic) and the fact that we in fact consider a past temporal logic that restricts situations involved in the fluents to ground situations. The proof of Theorem 2 uses standard logical rules and will be found in a longer version of this paper.

[8] We assume relational theories for classical databases.

3. *$\mathcal{D}_{ef}$ is the set of axioms and definitions for simple and complex event fluents which are expressed in a given event logic.*

Assume the notations of Theorem 1 above, and suppose that $seq\_ev^{FIFO}(r, t, e_1, e_2, s)$ and $seq\_ev^{LIFO}(r, t, e_1, e_2, s)$ denote event sequence fluents with the consumption modes FIFO and LIFO, respectively. Then we have the following result:

Theorem 2 *Suppose $\mathcal{D}$ is active relational theory with global consumption scope for the primitive event fluents. Then the following equivalences can be established:*

1. *First, Consumed-Last and Cumulative consumption modes are equivalent; i.e.,*

$$\begin{aligned}&\mathcal{D} \models seq\_ev^{F}(r, t, e_1, e_2, s) \textit{ iff}\\ &\mathcal{D} \models seq\_ev^{CL}(r, t, e_1, e_2, s) \textit{ iff}\\ &\mathcal{D} \models seq\_ev^{CUM}(r, t, e_1, e_2, s).\end{aligned}$$

2. *Non-Consumed-Last, LIFO, and FIFO consumption modes are equivalent; i.e.,*

$$\begin{aligned}&\mathcal{D} \models seq\_ev^{NL}(r, t, e_1, e_2, s) \textit{ iff}\\ &\mathcal{D} \models seq\_ev^{FIFO}(r, t, e_1, e_2, s) \textit{ iff}\\ &\mathcal{D} \models seq\_ev^{LIFO}(r, t, e_1, e_2, s).\end{aligned}$$

It is important to make clear what the equivalences above means. Intuitively, the logical equivalence of two consumption modes M_1 and M_2 means that any given sequence will occur at exactly the same situations under both M_1 and M_2. This ultimately leads to the same active behavior under both M_1 and M_2.

5 Related Work

The problem of specifying complex events in active databases has been recognized not to be trivial([3], [5], [4], [15]). Proposals for solving it can be classified in three groups. The first group uses regular expressions and context-free grammars to specify complex events (see, e.g., [5]). The second group uses graph-based methods to specify them ([3], [4]). Finally, the third group uses a logic-based approach (for good examples of this approach, see [5], [15]). Very often, these approaches are expressible in form of an *event algebra* ([16]). Using a metamodel for event algebras, Zimmer and Unland ([16]) provide a rich and systematic analysis of existing event algebras. Their metamodel, however, are not formal, contrary to the authors' claim. Our work, though, is most close to work reported in [2]. Here, the authors provide a logical framework for studying expressiveness of particular event algebras and decision problems related to these algebras. We obtain some of the results of [2], using a different framework, and we extend them on further consumption modes, especially First and Non-Consumed-Last, that were not considered in [2].

6 Conclusion

Many avenues lead out of our work. One is to identify more fragments of the situation calculus and study their expressiveness with respect to event logics. This meets one of the main motivations behind metamodels of [16], and would ultimately lead to a logic-based comparison of existing event algebras. A further avenue leads to studying tractable fragments which could serve as a basis for implementating ADBMSs. Finally, implementing the active relational theories is another avenue that is worth pursuing. We in fact provide implementable specifications that would allow one to check desirable properties of interest to one's intended system. The logical foundations for such an endeavor are laid down in [12].

Acknowledgments

We thank members of the Cognitive Robotics Group at the U. of T. for helpful discussions. Special thanks to Ray Reiter for insightful comments on what an event is. We also gratefully mention the financial support by NSERC, IRIS (Institute for Robotics and Intelligent Systems), and ITRC (Information Technology Research Centre of Ontario).

References

[1] M. Arenas and L. Bertossi. Hypothetical temporal queries in databases. In A. Borgida, V. Chaudhuri, and V. Staudt, editors, *Proceedings of the ACM SIGMOD/PODS 5th International Workshop on Knowledge Representation meets Databases (KRDB'98)*, pages 4.1–4.8, 1998. http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-10/.

[2] J. Bailey and S. Mikulás. Expressivemenss issues and decision problems for active database event queries. In *ICDT'2001*, pages 69–82, 2001.

[3] S. Chakravarthy and D. Mishra. An event specification language (snoop) for active databases and its detection. Technical Report UF-CIS-TR-91-23, University of Florida, 1991.

[4] S. Gatziu and K.R. Dittrich. Detecting composite events in active database systems using petri nets. In *Proceeding on Research Issues in Data Engineering, RIDE'94*, pages 2–9, 1994.

[5] N.H. Gehani, H.V. Jagadish, and O. Shmueli. Composite event specification in active databases: Model and implementation. In *Proceedings of the 18th VLDB Conference*, pages 327–338, Vancouver, 1992.

[6] I Kiringa. Simulation of advanced transaction models using golog. In *Proceedings of the 8th Biennial Workshop on Data Bases and Programming Languages (DBPL'01)*, 2001.

[7] I. Kiringa. Towards a theory of advanced transaction models in the situation calculus (extended abstract). In *Proceedings of the VLDB 8th International Workshop on Knowledge Representation Meets Databases (KRDB'01)*, 2001.

[8] H. Levesque, R. Reiter, Y. Lespérance, Fangzhen Lin, and R.B. Scherl. Golog: A logic programming language for dynamic domains. *J. of Logic Programming, Special Issue on Actions*, 31(1-3):59–83, 1997.

[9] J. McCarthy. Situations, actions and causal laws. Technical report, Stanford University, 1963.

[10] N.W. Paton. *Active Rules in Database Systems*. Springer Verlag, New York, 1999.

[11] R. Reiter. Towards a logical reconstruction of relational database theory. In M. Brodie, J. Mylopoulos, and J. Schmidt, editors, *On Conceptual Modelling*, pages 163–189, New-York, 1984. Springer Verlag.

[12] R. Reiter. *Knowledge in Action: Logical Foundations for Describing and Implementing Dynamical Systems*. MIT Press, Cambridge, 2001.

[13] A.P. Sistla and E.M. Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM*, 32:733–749, 1985.

[14] J. Widom and S. Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann Publishers, San Francisco, CA, 1996.

[15] C. Zaniolo. Active database rules with transaction-conscious stable-model semantics. In T.W. Ling and A.O. Mendelzon, editors, *Fourth International Conference on Deductive and Object-Oriented Databases*, pages 55–72, Berlin, 1995. Springer Verlag.

[16] D. Zimmer and R. Unland. On the semantics of complex events in active database managements systems. In *Proceedings of the International Conference on Data Engineering (ICDE)*, Sydney, 1999. Longer version at `http://www.cs.uni-essen.de/dawis/publications/`.