Composite Events in Chimera

Rosa Meo[1] Giuseppe Psaila[1] Stefano Ceri[2]

[1] Politecnico di Torino, Dipartimento di Automatica e Informatica
Corso Duca degli Abruzzi, 24 - I-10129 Torino, Italy
[2] Politecnico di Milano, Dipartimento di Elettronica e Informazione
Piazza L. Da Vinci, 32 - I-20133 Milano, Italy
rosimeo@polito.it, psaila/ceri@elet.polimi.it

Abstract. In this paper, we extend event types supported by Chimera, an active object-oriented database system. Chimera rules currently support disjunctive expressions of set-oriented, elementary event types; our proposal introduces instance-oriented event types, arbitrary boolean expressions (including negation), and precedence operators. Thus, we introduce a new event calculus, whose distinguishing feature is to support a minimal set of orthogonal operators which can be arbitrarily composed. We use event calculus to determine when rules are triggered; this is a change of each rule's internal status which makes it suitable for being considered by the rule selection mechanism.
The proposed extensions do not affect the way in which rules are processed after their triggering; therefore, this proposal is continuously evolving the syntax and semantics of Chimera in the dimension of event composition, without compromising its other semantic features. For this reason, we believe that the proposed approach can be similarly applied for extending the event language of other active database systems currently supporting simple events or their disjunction.

1 Introduction

Active database systems provide tight integration of Event-Condition-Action (ECA) rules within a database system. Such a tight integration is normally achieved by reusing database system components for implementing conditions (database predicates or queries) and actions (database manipulations, often embedded within a procedural component). In general, when a rule is selected for execution (or triggered), then its condition is evaluated (or considered), and if the condition is satisfied, then the action is immediately executed [3]. Thus, the condition-action (CA) components of an active database have a simple and uniform behavior, which is common to most active databases.

Instead, event type specification, evaluation, and coupling to conditions and actions have to be designed and implemented specifically for each active database system. Thus, it is not surprising that the notions of elementary event type, of event type composition, and of binding between event occurrences and the CA

[3] An exception is HiPAC [9] which supports several coupling modes between conditions and actions.

components are quite different in each active database, and such differences are responsible for most of the diversity of active databases.

Most active databases recognize just data manipulation operations (such as insert, delete, and update) as event types. The proposed SQL3 standard, currently under development by ANSI and ISO, associates to each rule just one event type; this can be considered as the simple extreme of a spectrum of solutions [17]. Most relational database products supporting active rules (called triggers) associate each of them to a disjunction of event types whose instances are relative to the same table [23]; this solution is also used by Starburst [24], Postgres [21], and Chimera, an active object-oriented database prototype developed at Politecnico di Milano in the context of the IDEA Esprit Project [4, 5]. More complex event calculus are supported by active database prototypes (see Section 1.1). In these approaches, rules are associated to event expressions which normally include generic boolean expressions, precedence operations, and explicit time references.

In all active rule systems, event instances cause rules to change an internal state; the corresponding state transition is called *triggering* of the rule. Once a rule is triggered, active rule systems react in several ways. When multiple rules are triggered at the same time, a *rule selection mechanism* determines which of them should be considered first; this mechanism may be influenced by priorities which are statically associated to rules. In addition, the rule selection may occurr *immediately* after the triggering operation or be *deferred* to some later point in transaction execution (such as the commit time). With immediate execution, it is possible to further identify the cases of rules executing *before*, *after*, or *instead of* the operation generating the triggering event occurrence. Finally, the triggering and execution of rules can be repeated for each tuple or object affected by an operation (*row-level granularity* in [17]) or instead relate to the overall set of tuples or objects manipulated by means of the same operation (*statement-level granularity* in [17]).

Due to all these alternatives, active rule systems present themselves with a variety of possible behaviors (a thorough comparative analysis of semantics supported by active rule systems is presented in [10]). In order to control the introduction of complex events in Chimera, and therefore the increase of semantic complexity due to this extension, we have strictly followed some design principles:

- We have defined the event calculus by means of a minimal set of orthogonal operators.
- The semantics of the event calculus is given simply by defining the conditions upon which rules having as event type a complex event calculus expression become triggered; detriggering occurs when a rule is selected for consideration and execution. No other state transitions characterize the internal state of each rule.
- The event calculus extension does not affect the way in which rules are processed after their triggering; therefore, this proposal continuously evolves the syntax and semantics of Chimera in the dimension of event type composition, without compromising its other semantic features.

We believe that these design principles are general and should drive the design of event calculus for active databases; therefore, we also believe that the proposed approach extends naturally to active database systems currently supporting simple event types or their disjunction.

The paper is organized as follows: Section 2 reports the current fundamental Chimera features; Section 3 introduces the proposed extension, while Section 4 formally gives its semantics; Section 5 deals with implementation issues; finally, Section 6 draws the conclusions.

1.1 Related work

There exist several Active Database Systems that have been provided with a language for event type composition; these languages are presented in [13], [7], [11], [19], [22]. The way all these proposals deal with composite event types is quite different depending on the particular systems; in fact, though they have similar sets of operators, different semantics have been proposed. In the rest of the section, we briefly discuss the most important proposals.

Ode [13] has a rich event language based on a small set of primitive operators. These operators deal with event occurrences in a set-oriented way, using set operations like intersection and complement: they produce subsets of the primitive event occurrence history, considered as an ordered set based on the event occurrence time-stamps. For example, event conjunction is the set of event occurrences that satisfy both component event types (and it produces a not null result provided that the two event occurrence sets corresponding to the two operands have at least one common element); event negation is the complement with respect to the whole history; *relative* of an event type A with respect to type B is the set of occurrences of type B subsequent to the first occurrence of type A. Other operators, like event disjunction, temporal precedence (*prior*), strict sequence (*sequence*), etc., are derived from the primitive operators. The user is allowed to specify conditions on event properties directly in the composition expressions, i.e. in the event part of the rule. Since the expressive power is that of regular expressions, composite events are checked by means of a finite state automata.

HiPAC [8] makes available *data manipulations* events, *clock* events and *external* events. Clock events can be specified as *absolute*, *relative* and *periodic*. Composite event types are defined with the use of the following operands: *disjunction*, *sequence* (temporal precedence of event signals) and *closure* (event signals occurred one or more times).

Snoop [7] interprets an event E as a boolean function defined on the time domain that is true at time t if an event occurrence of that event type occurs at time t. Event conjunction and disjunction are obtained by the boolean algebra applied on their operands. Negation of an event E is defined as the absence of an occurrence of E in a closed interval determined by two events E_1 and E_2. While the *aperiodic* operator determines the presence of all the occurrences of an event E between two subsequent occorrences of E_1 and E_2, the *periodic* operator is equivalent to a periodic event generator: given a time period t_p, it is true at

instants separated each other by t_p, starting from an occurrence of an event E_1 and stopping at the subsequent occurrence of an event E_2. The *cumulative* versions of these two last operators are defined as "accumulating" respectively occurrences of E and time instants. Depending on the application, it is possible to define different *contexts* in order to let a rule be triggered in correspondance of either all the possible combinations of primitive event occurrences matching the event expression or only some of them.

Samos [11] has a rather rich language as well, which provides the usual event disjunction, conjunction and *sequence* (ordered conjunction). A *Times* operator returns the point in time when the n-th occurrence of a specified event is observed in a given time interval. The negation is defined as the absence of any occurrence of an event type from a given time interval, and occurs at its end point. A star (*) operator returns the first occurrence of a given event type, regardless of the number of occurrences. Samos allows information passing from the event to the condition part of the rule by means of *parameters* like the identifier of the transaction in which a given event occurred (*c_tid*), or the point in time of the event occurrence (*occ_point*). Composite event parameters are derived: disjunction and star (*) receive the parameters of the component event occurrences, conjunction and *Times* their union. A keyword *same* specifies that the component events of a composition must have the same parameters.

The Reflex system [19] is an active database system designed for knowledge management applications. Its event algebra provides operators similar to those of Samos. These operators can be classified as logical (*and*, *or*, *xor* and *not*) or temporal (*precedes*, *succeeds*, *at*, *between*, *within* time-spec, *every* time-spec, etc..)

IFO_2 [22] is a conceptual model designed to capture both the structural and behavioural aspects of the modeled reality. The behavioural nodel is based on the notion of event, that represents either a fact of the modelled system which occurs in a spontaneous manner (in the case of external or temporal events) or is generated by the application. The event constructors are *composition* (conjunction), *sequence* (temporal precedence), *grouping* (collection of events of the same type) and *union* (disjunction). When a IFO_2 schema is defined, it is possible to translate it into a set of *ECA* rules by means of an ad hoc algorithm.

2 Introduction to Chimera

Chimera is a novel active, object-oriented and deductive database system; the main goal of its design was the definition of a clear semantics, especially for those aspects concerning active rules, such as rule execution, coupling modes, triggering.

Chimera active rules (also called *triggers*) follow the ECA (Event-Condition-Action) paradigm. Each trigger is defined on a set of *triggering events*, and it becomes active if *any* of its triggering events occurs. The Chimera event language was designed to consider only *internal events*, i.e. events generated by updates or queries on the database, like *create*, *modify*, *delete*, *generalize*, *specialize*, *select*,

etc.. In particular, a rule is defined either as *targeted* or *untargeted*: if targeted to a class, only events regarding that class are considered for triggering, otherwise events regarding any class in the database can appear in the event part of the rule.

The condition part is a logical formula that may perform a query on the database; its evaluation is called *consideration*. Depending on the success of this evaluation, the action part is executed coupled with the condition part.

Chimera does not permit binding transfer from the event section to the condition section because of the set-oriented approach; nevertheless, it is important for conditions to obtain objects affected by occurred events. Thus, a condition may include *event formulas*, particular formulas that query the event base and create bindings to the objects affected by a specified set of event types. Two predicates are available to write event formulas: the *occurred* predicate and the *holds* predicate. The former one extracts all the objects affected by the specified event types; the latter considers event composition.

A rule is *triggered* as soon as one of the triggering events arises, and it is no longer taken into account for triggering, until it has been considered. The triggering mechanism checks for new triggered rules immediately after a non interruptable execution block (either a user instruction sequence, called *transaction line*, or a rule action).

Based on the Event-Condition (EC) *coupling mode* chosen by the user, the rule behaves differently: if the rule is defined as *immediate*, the consideration is performed as soon as possible after the termination of the non interruptable block that generated the triggering event occurrence; if the rule is *deferred*, it is suspended until the *commit* command is given.

After the triggering mechanism has checked for new triggered rules, it chooses a rule to be considered and possibly executed, if there is any triggered rule; the choice is made based on a partial order derived from rule priorities provided by the user. Notice that after the consideration and possibly the execution of the rule, it is detriggered and it can be triggered again only by new event occurrences, because events occurred before the consideration loose the capability of triggering the rule.

The user can influence the behaviour of the rule specifying the Event Consumption mode as either *consuming* or *preserving*: in the former case, only event occurrences more recent than the last consideration of the trigger are accessible to event formulas; in the latter, all the events occurred since the beginning of the transaction are available.

Example The following rule reacts to the creation of stock items, to check whether the quantity exceeds the maximum quantity admitted for that item.

```
define immediate checkStockQty for stock
       events: create
       condition: stock(S), occurred(create, S),
                  S.quantity>S.max_quantity
```

```
        action: modify(stock.quantity, S, S.max_quantity)
end;
```

The rule, called *checkStockQty* is defined with *immediate* EC coupling mode and is targeted to the *stock* class. The event part indicates that the rule is triggered when a *create* event on class *stock* occurs. The condition is structured as follows: a variable `S` is defined on class *stock*; the *occurred* predicate binds the objects affected by the creation to that variable and finally the constraint is checked. If there is some object that violates the constraint, then the action changes its quantity setting it to the maximum quantity for that object. Note that the rule is executed in a set-oriented way, so all the objects created and not checked yet by the rule are processed together in a single rule execution.

3 Extending Chimera with Composite Events

Our extension of Chimera with composite event types moves from the currently available features in order to preserve the characterizing aspects of this system. In particular, the introduction of an event calculus language should change neither the triggering/detriggering semantics, nor the processing of triggered rules, in particular with respect to *EC compling modes* and *event consumption.*

A composite event is an event expression obtained from primitive event types by means of a set of operators, such as *conjunction*, *disjunction*, *negation* and temporal *precedence*. These operators are divided in set-oriented and instance-oriented operators: in the former case, we consider the occurrence of a combination of event types independently of the affected objects; in the latter, the specified combination must occur on the same object. They are reported in Figure 1, listed in decreasing priority order: set-oriented operators have lower priority than instance-oriented ones, and conjunction and precedence operators have the same priority. A complete introduction to the event calculus language follows in next Sections 3.1 and 3.2. While designing the language, we moved on three othogonal dimensions, as depicted in Figure 2: due to the *boolean dimension*, we introduced operators such as *conjunction*, *disjunction* and *negation*; due to the *granularity dimension*, these operators are divided in *instance-oriented* and *set-oriented*; due to the *temporal dimension* we introduced two *precedence* operators, one instance-oriented and the other set-oriented.

In the following two Sections, we introduce the set of operators. For each event expression built by means of each operator, we indicate whether the event has occurred (we say that the event is *active*) and we indicate the most recent time when the event has occurred (called its *activation time-stamp*). We make use of some sample event expressions based on classes *stock*, describing stock products, and *show*, indicating products on shelves in a sale-room.

3.1 Set-Oriented Operators

A *primitive event* occurs when an occurrence of that event type arises, independently of the object affected by it. For instance, let us imagine that two

occurrences of the event `create(stock)` arise at time t_1 and t_2. At time $t < t_1$ the event is not active; at time $t_1 \leq t < t_2$ the event is active and its activation time-stamp is t_1; finally, at time $t_2 \leq t$ the event is active and its activation time-stamp is t_2.

The first version of Chimera already provided the disjunction among primitive events, that was described by a list of primitive event types separated by commas. We keep the same notation, extending its application to generic event expressions. Intuitively the *disjunction* $\mathcal{E}_1$`,`$\mathcal{E}_2$ of two event expressions arises as soon as one of the component events becomes active. To be more precise, we say that at a certain time t the disjunction is active if at least one of the component events is active. If only one of the component events is active, its activation time-stamp becomes the activation time-stamp of the disjunction; if both the components events are active, the highest activation time-stamp of them is assumed to be the activation time-stamp of the disjunction.

For instance, as an example let us consider the sample event expression `create(stock), modify(stock.quantity)`, two occurrences of the primitive event `create(stock)` at times t_1 and t_3, and one occurrence of the primitive event `modify(stock.quantity)` at time t_2, with $t_1 < t_2 < t_3$. At time $t < t_1$ the disjunction event is not active; at time $t_1 \leq t < t_2$ the disjunction is active and its activation time-stamp is t_1; at time $t_2 \leq t < t_3$, the disjunction is active and its activation time-stamp is now t_2; finally, at time $t \geq t_3$ the disjunction is active and its activation time-stamp is now t_3.

When we consider the *conjunction* $\mathcal{E}_1$`+`$\mathcal{E}_2$ of two events, it is intuitive that it is active when both of the component events are active. If so, the activation time-stamp is the highest of the activation time-stamps of the component events.

For instance, as an example let us consider the sample event expression `create(stock) + modify(stock.quantity)`, two occurrences of the primitive event `create(stock)` at times t_1 and t_3, and one occurrence of the primitive event `modify(stock.quantity)` at time t_2, with $t_1 < t_2 < t_3$. At time $t < t_1$ the conjunction event is not active; at time $t_1 \leq t < t_2$ the conjunction is still not active; at time $t_2 \leq t < t_3$, the conjunction is active and its activation time-stamp is t_2; finally, at time $t \geq t_3$ the activation time-stamp is now t_3.

In complex applications it is often necessary to consider the absence of an event, i.e. one would like to check for the absence of occurrences of an event.

We think that a *negation* event `-` $\mathcal{E}$ is active when the *negated* event (also called component event) is not active; in particular, if there are no occurrences

	Instance-Oriented	*Set-Oriented*
Negation	`-=`	`-`
Conjunction	`+=`	`+`
Precedence	`<=`	`<`
Disjunction	`,=`	`,`

Fig. 1. Composition Operators Set.

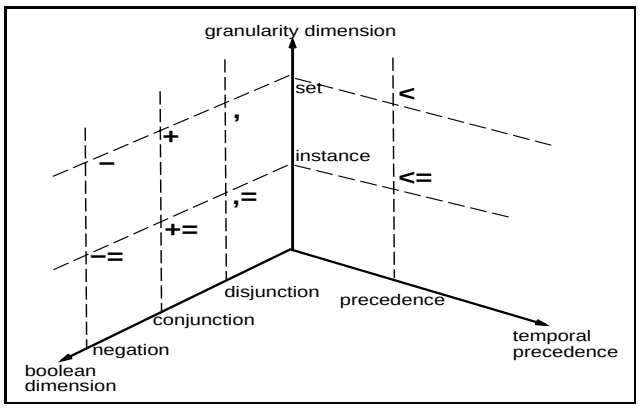

Fig. 2. Event operators dimensions

of the negated event at time t, the activation time-stamp is the current time.

For instance, let us consider the first occurrence of the event `create(stock)` at time t_1 and its negation, `-create(stock)`. At time $t \geq t_1$, since the event `create(stock)` is active, the negation is not active; at time $t < t_1$, since `create(stock)` is not active, the negation is active and its activation time-stamp is t, because it is occurring at time t.

Similarly to the conjunction, the *precedence* $\mathcal{E}_1$<$\mathcal{E}_2$ of two event expressions is active provided that both the component events are active; moreover, the first component event must become active earlier than the second one.

For instance, as an example let us consider the sample event expression `create(stock) < modify(stock.quantity)`, two occurrences of the primitive event `create(stock)` at times t_1 and t_3, and one occurrence of the primitive event `modify(stock.quantity)` at time t_2, with $t_1 < t_2 < t_3$. At time $t < t_1$ the precedence event is not active; at time $t_1 \leq t < t_2$ the precedence is still not active; at time $t_2 \leq t < t_3$, the precedence is active and its activation time-stamp is t_2; finally, at time $t \geq t_3$ the precedence is active and its activation time-stamp still remains at t_2, because the second creation has time-stamp greater than that of the last modification.

We are able to write any complex set-oriented event expression, e.g.

```
modify(show.quantity) +
   ( -( create(stockOrder) < modify(stockOrder.del_quantity)) ,
     ( modify(stock.min_quantity) < modify(stock.quantity) ) )
```

which is active if there is a modification of the product quantity on a *shelf*, and there is not a creation of a *stock order* followed by a modification of the delivered quantity for a *stock order*, or there is a modification of the minimum quantity for a *stock* followed by a modification of the quantity for a *stock*.

3.2 Instance-Oriented Operators

Instance-Oriented operators are useful to catch the occurrence of composite events on the same object. For this reason, instance-oriented operators have higher priority than set-oriented ones, and they cannot be applied to event sub-expressions obtained by means of set-oriented operators.

In contrast, an event expression obtained using instance-oriented operators can appear as an operand of a set-oriented operator; in fact, it is very intuitive to pass from the instance-oriented to the set-oriented level, as we will show later.

A *primitive event* occurs on an object O when a new occurrence of that event type arises and affects O. As in the set-oriented case, at time t the following situations are possible: no event occurrences of that type have arisen yet on O, so the primitive event is *not active* for O; at least one occurrence of that type has arisen on O, then the primitive event is *active* for O and the *activation time-stamp* is that of the more recent occurrence. For instance, let us imagine that two occurrences of the event `create(stock)` arise at time t_1 and t_2 on the objects O_1 and O_2 respectively. At time $t < t_1$ the event is not active for both the objects; at time $t_1 \leq t < t_2$ the event is active only for O_1 and its activation time-stamp is t_1; finally, at time $t_2 \leq t$ the event is still active for O_1 with activation time-stamp t_1, but it becomes active for O_2 too and its activation time-stamp is t_2.

The *instance-oriented conjunction* $\mathcal{E}_1$`+=`$\mathcal{E}_2$ of two events on the same object O, is active when both the component events are active for O. The activation time-stamp for O is the highest of the activation time-stamps for the component events. For instance, `create(stock) += modify(stock.quantity)` is an instance-oriented conjunction that becomes active for a *stock* object O when O has been created and its quantity has been changed. When used in a set-oriented expression, an instance-oriented conjunction is active if there is at least one object affected by the two component event expressions. For instance, consider the expression

```
modify(show.quantity) + (create(stock) += modify(stock.quantity))
```

which is active when a change of a shown product quantity occurs and at least a *stock* object has been created and its quantity modified.

The *instance-oriented disjunction* $\mathcal{E}_1$`,=`$\mathcal{E}_2$ of two event expressions on an object O intuitively arises as soon as one of the component events becomes active for O. Precisely, at a certain time t the disjunction is active for O if at least one of the component events is active for O. If only one of the component events is active for O, its activation time-stamp becomes the activation time-stamp of the disjunction; if both the components events are active for O, the highest activation time-stamp of them is assumed to be the activation time-stamp of the disjunction. Consider the expression `create(stock),= modify(stock.quantity)` as an example of instance-oriented disjunction, two occurrences of the primitive event `create(stock)` at times t_1 and t_3 on objects O_1 and O_3 respectively, and two occurrences of the event `modify(stock.quantity)` at time t_2 on objects O_1 and O_2 respectively, with $t_1 < t_2 < t_3$. At time $t < t_1$ the disjunction event is

not active for all the three mentioned objects; at time $t_1 \leq t < t_2$ the disjunction is active for O_1 with activation time-stamp t_1 and still not active for O_2 and O_3; at time $t_2 \leq t < t_3$, the disjunction is still active for O_1 with activation time-stamp t_1 but is now active for O_2 with activation time-stamp t_2; finally, at time $t \geq t_3$ the disjunction is now active also for O_3 with activation time-stamp t_3. When used in a set-oriented expression, an instance-oriented disjunction is active if there is at least one object affected by the disjunction of the component event expressions. For instance, consider the expressions:

```
modify(show.quantity) + (create(stock) ,= modify(stock.quantity))
modify(show.quantity) + (create(stock) , modify(stock.quantity))
modify(show.quantity) +
 (create(stock) +=
  (modify(stock.min_quantity) ,= modify(stock.quantity)))
```

The first one is active when a change of a shown product quantity occurs and a *stock* object has been created or its quantity modified. Observe that with such a use the effect is the same as the second expression, that is active when a change of a shown product quantity occurs and there are a creation of a *stock* object or a modification of the quantity for a *stock* object, the two objects being possibly different; in fact, the instance-oriented disjunction operator has been introduced to be used in instance-oriented event expressions, based on a operator set that we want to be orthogonal w.r.t. set-oriented operator set. The third expression clarifies this concept, because it is active when a change of a shown product quantity occurs and there is a creation of a *stock* objecy on which either a modification of the minimum quantity or a modification of the quantity occur.

The *instance-oriented negation* `-=` $\mathcal{E}$ expresses the absence of occurrences of an event type for an object O: it is active when the *negated* event is not active for O and the activation time-stamp is the current time. For instance, let us consider two occurrences of the event `create(stock)` at time t_1 and t_2 affecting O_1 and O_2 respectively, and the negation event `-=create(stock)`. At time $t < t_1$ the negation is active for both O_1 and O_2, with activation time-stamp t for both; at time $t_1 \leq t < t_2$, since the event `create(stock)` is active for O_1 but not for O_2, the negation is not active for O_1 but is still active for O_2 with activation time-stamp t; finally, at time $t_2 \leq t$, since `create(stock)` is active for both, the negation is not active for both O_1 and O_2. It is easy to think of the use of an instance-oriented negation in the following intuitive way: it is active if there is no object which the instance-oriented negation is active for, otherwise it is not active. Notice that if the `-=` operator is applied to elementary event types, using it in a set-oriented expression leads to the same result as the set-oriented version; things change when it is applied to more complex instance-oriented expressions. For instance, consider the expressions:

```
modify(show.quantity) + -=(create(stock)+=modify(stock.quantity))
modify(show.quantity) + -(create(stock) + modify(stock.quantity))
```

The first one is active when a change of a shown product quantity occurs and no *stock* object has been created and its quantity modified. Instead, the second one is active when a change of a shown product quantity occurs and there is neither

a creation of a *stock* object nor a modification of the quantity for a *stock* object, the two objects being possibly different.

Similarly to the conjunction, the *instance-oriented precedence* $\mathcal{E}_1$<=$\mathcal{E}_2$ of two events is active when both the component events are active on the same object O, the first one becoming active earlier than the second one. For instance, let us consider `modify( stock.min_quantity) <= modify( stock.quantity)`, two occurrences of the event `modify(stock.min_quantity)` at times t_1 and t_3 on the same object O_1, and one occurrence of the event `modify(stock.quantity)` at time t_2 again on the object O_1, with $t_1 < t_2 < t_3$. At time $t < t_1$ the precedence event for O_1 is not active; at time $t_1 \leq t < t_2$ the precedence is still not active for O_1; at time $t_2 \leq t < t_3$, the precedence is active for O_1 and its activation time-stamp is t_2; finally, at time $t \geq t_3$ the precedence is active for O_1 and its activation time-stamp is still t_2. When used in a set-oriented expression, an instance-oriented precedence is active if there is at least one object affected by the sequence of the two component event expressions. For instance, consider the expressions:

```
modify(show.quantity) + (create(stock) <= modify(stock.quantity))
modify(show.quantity) + (create(stock) < modify(stock.quantity))
```

The first one is active when a change of a shown product quantity occurs and at least a *stock* object has been created and later its quantity modified. Instead, the second one is active when a change of a shown product quantity occurs and there is a creation of a *stock* object followed by a modification of the quantity for a *stock* object, the two objects being possibly different.

3.3 Event Formulas

As introduced in Sections 3, event formulas (see Section 2) are extended in consequence of the introduction of the event language.

Event expressions. The *occurred* predicate now supports event expressions limited to *instance-oriented* operators. This is due to the semantics of the predicate: it returns all the objects affected by the specified event expression [4]. For example:

```
occurred( create(stock) <= modify(stock.quantity) , X )
```

binds all the objects created whose attribute `quantity` has been modified to variable `X`. Depending on the *consumption mode* selected for the rule, the above formula retrieves either all the objects affected by that particular combination

[4] Chimera supports also a predicate *holds* which composes event types. However, there is no need of such predicate in the new Chimera extended with event calculus, since event composition can be explicitily evaluated by the calculus. For instance, net effect for the creation operation in presence of sequences of modifications and deletions is given by the following event formula:

```
create(class) ≡
 (create(class) ,=  (create(class) <= modify(class.attr))) +=
 -=(create(class) <= delete(class))
```

of event types since the beginning of the transaction (*preserving* rule) or only those affected since the last consideration of the rule (*consuming* rule). Observe that this is exactly the same semantics of Chimera without composite events, reviewed in Section 2.

Occurrence time-stamp. This new predicate is similar to the *occurred* predicate but it provides the time-stamp of the specified composite event occurrences as well. For example:

```
at( create(stock) <= modify(stock.quantity) , X, T )
```

where T is a variable defined on type *time*.

Its semantics is defined as follows: given an object `X`, `T` assumes all the time-stamps, in the observed time interval, at which an occurrence of the specified event expression arises for that object. In the above example, if the creation of a stock object is followed by two updates of its `quantity` attribute, the specified composite event occurs twice, exactly when the two updates occur.

The observed time interval depends on the consumption mode selected for the rule: it can range either from the beginning of the transaction to the current time (*preserving* rules), or from the last consideration of the rule to the current time (*consuming* rule).

4 Formal Semantics

This Section is organized as follows: at first, we describe our approach to the definition of event calculus; then we precisely define our model of *Event Base*, on which the definitions presented later are based; then, we give the formal semantics for both *set-oriented* and *instance-oriented* operators; finally, triggering semantics is formulated in formal way.

Composite event semantics The main goal of our work is to provide the event language with a semantics that preserves boolean properties, such as De Morgan rules, when time properties associated to event occurrences (their time-stamp) are considered. In fact, event occurrences exist because they are generated at a certain time instant, thus that aspect should be always taken into account when event expressions are evaluated.

Event expressions are used in the event part of the rule and possibly in event formulas in the condition part. So the event occurrence determine whether the rule is triggered or not.

The main idea is the following: when a portion of the event base EB (the log of all events occurred since the beginning of the transaction, see Section 4.1) is investigated, for each primitive event type we construct a function dependent on time t, called the *time-stamp of the more recent event occurrence* in the investigated portion of EB, indicated with ts. The ts function of an event type is constructed on the basis of the positive time-stamp of the last occurrence of the event type, if an event occurrence exists in the investigated portion of EB. Otherwise, ts calculated in t is set to a negative value, equal to $-t$.

EID	event-type	OID	time-stamp
e_1	$< create,\ stock >$	o_1	t_1
e_2	$< create,\ stock >$	o_2	t_2
e_3	$< create,\ order >$	o_3	t_3
e_4	$< create,\ notFilledOrder >$	o_3	t_3
e_5	$< modify,\ stock,\ quantity >$	o_1	t_4
e_6	$< modify,\ stock,\ quantity >$	o_2	t_4
e_7	$< delete,\ stock >$	o_1	t_5

Fig. 3. Example of EB.

Thus, the sign of the ts function of an event type states whether an occurrence of that event type exists in the portion of EB relevant for rule triggering: if positive, an event type occurrence exists; if negative, otherwise. Consequently, it is sufficient to determine an instant t in which function ts is positive to solve rule triggering.

When dealing with negation, the intuition is that ts function of a negation event calculated at time t has the opposite value of ts of the negated component event, for each time value. In fact if an occurrence of an event type does not exist in the portion of EB relevant for rule triggering (i.e. all event occurrences more recent than the last consideration of the rule), from the instant of the last consideration of the rule, ts value of the negated event at time t, is the time value t. It comes that ts functions of primitive event types are calculated by a simple lookup into a portion of EB.

From these basic ts functions, our event calculus algebraically derives ts functions for event expressions from ts functions associated to its primitive components. As already said, these expressions are obtained applaying arbitrarily boolean operators and precedence operator to primitive event types. Derived ts functions associated to event expressions have the same properties of ts functions for primitive event types.

4.1 The Event Base

The Event Base (EB) is the log containing all the event occurrences since the beginning of the transaction. In this paper we model the EB as a table having the structure depicted in Figure 3.

Each row contains an event occurrence, characterized by its unique identifier (EID), the event type, the Object Identifier (OID) of the object affected by the event occurrence, and the time-stamp of the time instant the event occurred at. The event type is described by the name of the command that changed the object state, possibly followed by the object class name and an attribute name. In the following, we refer to the EID of a generic event occurrence as e.

Given an event occurrence e, we can define a set of useful functions returning properties of e stored in the EB. Figure 4 contains examples of the defined functions; derived from the EB state of Figure 3.

$type: e_1 \rightarrow < create, stock >$	$obj: e_4 \rightarrow o_3$
$type: e_5 \rightarrow < modify, stock, quantity >$	$obj: e_5 \rightarrow o_1$
$type: e_6 \rightarrow < delete, stock >$	$obj: e_6 \rightarrow o_2$
$timestamp: e_2 \rightarrow t_2$	$eventonclass: e_1 \rightarrow stock$
$timestamp: e_3 \rightarrow t_3$	$eventonclass: e_5 \rightarrow stock$
$timestamp: e_4 \rightarrow t_3$	

Fig. 4. Examples of event attribute matches on events in EB.

type $type: EID \rightarrow eventtype$

This function matches each event occurence to its event type.

obj $obj: EID \rightarrow OID$

This relation matches each event occurrence to the object whose state has been modified by that event.

timestamp $timestamp: EID \rightarrow time$

This function matches each event occurrence to its time-stamp.

eventonclass $eventonclass: EID \rightarrow classname$

This function matches each event occurrence to the class to which the object affected by the event occurrence belongs. Note that this piece of information is part of the *event-type* attribute.

4.2 Set-Oriented Case

The definition of ts of a primitive event type $\mathcal{E}$ at time t is:

$$ts(\mathcal{E}, t) \stackrel{\text{def}}{=} \begin{cases} -t \;\; if\ \forall t'\ (t' \leq t \wedge \nexists e \in R\ (\\ \qquad type(e) = \mathcal{E} \wedge timestamp(e) = t')) \\ t_E \;\; otherwise, where \\ \qquad t_E = max\{t'(t' \leq t \;\wedge\; \exists e \in R(type(e) = \mathcal{E} \\ \qquad\quad \wedge\ timestamp(e) = t'))\} \end{cases}$$

where R is the set of event occurrences to which the event calculus applies.

We also introduce the function u(t): $u(t) \stackrel{\text{def}}{=} 0$ if $t \geq 0$, $u(t) \stackrel{\text{def}}{=} 1$ if $t \geq 0$.

From the above definitions, the presence of an event occurrence in R at time t, is expressed by the logical predicate $occ(\mathcal{E}, t)$ which is $true$ if $u(\ ts(\mathcal{E}, t)) = 1$, $false$ otherwise.

As already said informally, the semantics of *negation* is $ts(\text{-}\mathcal{E}, t) \stackrel{\text{def}}{=} -ts(\mathcal{E}, t)$.

Semantics of the other set-oriented operators is given in two steps: at first, we give a precise definition in logical style; second, that definition is translated into an algebraic equivalent expression that can be used for the evaluation of the ts function associated to the overall event expression.

LogicalStyleSemantics

1) $ts(\mathcal{A}+\mathcal{B},t) \stackrel{\text{def}}{=} \begin{cases} min\{ts(\mathcal{A},t),ts(\mathcal{B},t)\} \ if \ \neg occ(\mathcal{A},t) \vee \neg occ(\mathcal{B},t) \\ max\{ts(\mathcal{A},t),ts(\mathcal{B},t)\} \ if \ occ(\mathcal{A},t) \wedge occ(\mathcal{B},t) \end{cases}$

2) $ts((\mathcal{A},\mathcal{B}),t) \stackrel{\text{def}}{=} \begin{cases} min\{ts(\mathcal{A},t),ts(\mathcal{B},t)\} \ if \ \neg occ(\mathcal{A},t) \wedge \neg occ(\neg\mathcal{B},t) \\ max\{ts(\mathcal{A},t),ts(\mathcal{B},t)\} \ if \ occ(\mathcal{A},t) \vee occ(\mathcal{B},t) \end{cases}$

3) $ts((\mathcal{A}<\mathcal{B}),t) \stackrel{\text{def}}{=} \begin{cases} -t \quad if \ \neg occ(\mathcal{A},t) \vee \neg occ(\mathcal{B},t) \\ \qquad \vee \ occ(\mathcal{A},t) \wedge occ(\mathcal{B},t) \\ \qquad \wedge \ ts[\mathcal{A},ts(\mathcal{B},t)] < 0 \\ ts(\mathcal{B},t) \ if \ occ(\mathcal{A},t) \wedge occ(\mathcal{B},t) \\ \qquad \wedge \ ts[\mathcal{A},ts(\mathcal{B},t)] \geq 0 \end{cases}$

AlgebraicSemantics

1) $ts(\mathcal{A}+\mathcal{B},t) = min\{ts(A,t),ts(B,t)\}\ [1 - \ u(\ ts(\mathcal{A},t))\ u(\ ts(\mathcal{B},t))]+ \\ max\{ts(A,t),ts(B,t)\}\ [u(\ ts(\mathcal{A},t))\ u(\ ts(\mathcal{B},t)))]$

2) $ts((\mathcal{A},\mathcal{B}),t) = max\{ts(\mathcal{A},t),ts(\mathcal{B},t)\}\ [1 - u(-ts(\mathcal{A},t))\ u(-ts(\mathcal{B},t))]+ \\ min\{ts(\mathcal{A},t),ts(\mathcal{B},t)\}\ [u(-ts(\mathcal{A},t))\ u(-ts(\mathcal{B},t))]$

3) $ts((\mathcal{A}<\mathcal{B}),t) = -t[1 - u(\ ts(\mathcal{B},t))\ u(\ ts(\mathcal{A},ts(\mathcal{B},t)))]+ \\ ts(\mathcal{B},t)[u(\ ts(\mathcal{B},t))\ u(\ ts(\mathcal{A},ts(\mathcal{B},t)))]$

It is possible to show that several properties holds, like the De Morgan property that $ts(\text{-}((\text{-}\mathcal{A})+(\text{-}\mathcal{B})),t)$ is equivalent to $ts((\mathcal{A},\mathcal{B}),t)$.

$$
\begin{aligned}
& ts(\text{-}((\text{-}\mathcal{A})+(\text{-}\mathcal{B})),t) = \\
& = -min\{ts(\text{-}\mathcal{A},t),ts(\text{-}\mathcal{B},t)\}\ [1 - \ u(\ ts(\text{-}\mathcal{A},t))\ u(\ ts(\text{-}\mathcal{B},t))]+ \\
& \quad max\{ts(\text{-}\mathcal{A},t),ts(\text{-}\mathcal{B},t)\}\ [u(ts(\text{-}\mathcal{A},t))\ u(\ ts(\text{-}\mathcal{B},t)))] = \\
& = max\{ts(\mathcal{A},t),ts(\mathcal{B},t)\}\ [1 - u(-ts(\mathcal{A},t))\ u(-ts(\mathcal{B},t))]+ \\
& \quad min\{ts(\mathcal{A},t),ts(\mathcal{B},t)\}\ [u(-ts(\mathcal{A},t))\ u(-ts(\mathcal{B},t))] = \\
& = ts((\mathcal{A},\mathcal{B}),t)
\end{aligned}
$$

□

A graphical proof of this property is shown in Figure 5: for a set of event occurrences (of type A, B, and C, where event type C is not involved in the expression), it shows ts functions for both primitive and complex event expressions used in De Morgan proof.

The dual De Morgan property $ts((\mathcal{A}+\mathcal{B}),t) = ts(\text{-}(\text{-}\mathcal{A},\text{-}\mathcal{B}),t)$ and the following properties can be proved analogously.

$\mathcal{E}_1+\mathcal{E}_2 = \mathcal{E}_2+\mathcal{E}_1$	$(\mathcal{E}_1+\mathcal{E}_2)<\mathcal{E}_3 = (\mathcal{E}_1<\mathcal{E}_3)+(\mathcal{E}_2<\mathcal{E}_3)$
$\mathcal{E}_1,\mathcal{E}_2 = \mathcal{E}_2,\mathcal{E}_1$	$(\mathcal{E}_1,\mathcal{E}_2)<\mathcal{E}_3 = (\mathcal{E}_1<\mathcal{E}_3),(\mathcal{E}_2<\mathcal{E}_3)$
$(\mathcal{E}_1+\mathcal{E}_2)+\mathcal{E}_3 = \mathcal{E}_1+(\mathcal{E}_2+\mathcal{E}_3)$	$\mathcal{E}_1<(\mathcal{E}_2+\mathcal{E}_3) = (\mathcal{E}_1<\mathcal{E}_2),(\mathcal{E}_1<\mathcal{E}_3)$
$(\mathcal{E}_1,\mathcal{E}_2),\mathcal{E}_3 = \mathcal{E}_1,(\mathcal{E}_2,\mathcal{E}_3)$	$\mathcal{E}_1<(\mathcal{E}_2,\mathcal{E}_3) = (\mathcal{E}_1<\mathcal{E}_2)+(\mathcal{E}_1<\mathcal{E}_3)$
$\mathcal{E}_1+(\mathcal{E}_2,\mathcal{E}_3)) = \mathcal{E}_1+\mathcal{E}_2,\mathcal{E}_1+\mathcal{E}_3$	$\mathcal{E}_1<\mathcal{E}_2<\mathcal{E}_3 = (\mathcal{E}_1<\mathcal{E}_2)<\mathcal{E}_3$

4.3 Instance-Oriented Case

Instance-Oriented operators are useful to catch the occurrence of composite events on the same object. For this reason, instance-oriented operators have higher priority than set-oriented ones, and they cannot be applied to event subexpressions obtained by means of set-oriented operators.

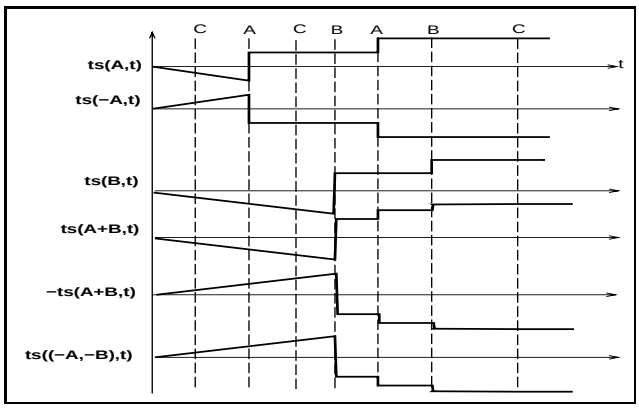

Fig. 5. Examples of *ts* functions with event expression

The exposition will follow a schema similar to that of Section 4.2: after the introduction of basic definitions and instance-oriented composition operators, we show how instance-oriented event expressions are evaluated inside set-oriented expressions.

In the *instance-oriented* case we make use of *ots* functions, which are very similar to *ts* functions, except for the fact that they refer to a single object.

$$ots(\mathcal{E},t,oid) \stackrel{\text{def}}{=} \begin{cases} -t & if\ \forall t'(t' \leq t \wedge \nexists e \in R(type(e) = \mathcal{E} \\ & \quad \wedge timestamp(e) = t' \wedge obj(e) = oid)) \\ t_E & otherwise, where \\ & t_E = max\{t'(t' \leq t\ \wedge \exists e \in R(type(e) = \mathcal{E} \\ & \quad \wedge\ timestamp(e) = t' \wedge obj(e) = oid))\} \end{cases} $$

where R is the set of event occurrences to which the event calculus applies. As in the set-oriented case, it is $oocc(\mathcal{E},t,oid) = true$ if $u(\ ots(\mathcal{E},t),oid) = 1$, $false$ otherwise.

Conjunction : LogicalStyleSemantics

$$ots(\mathcal{A}\text{+=}\mathcal{B},t,oid) \stackrel{\text{def}}{=} \begin{cases} min\{ots(\mathcal{A},t,oid),ots(\mathcal{B},t,oid)\}\ if\ \neg oocc(\mathcal{A},t,oid) \vee \neg oocc(\mathcal{B},t,oid) \\ max\{ots(\mathcal{A},t,oid),ots(\mathcal{B},t,oid)\}\ if\ oocc(\mathcal{A},t,oid) \wedge oocc(\mathcal{B},t,oid) \end{cases} $$

Conjunction : AlgebraicSemantics

$$\begin{aligned} &ots(\mathcal{A}\text{+=}\mathcal{B},t,oid) = \\ &\quad min\{ots(A,t,oid),ots(B,t,oid)\}\ [1 - u(ots(\mathcal{A},t,oid))u(ots(\mathcal{B},t,oid))] + \\ &\quad max\{ots(A,t,oid),ots(B,t,oid)\}\ [u(ots(\mathcal{A},t,oid))u(ots(\mathcal{B},t,oid))] \end{aligned}$$

The disjunction, negation and precedence operators are similarily extended to the instance-oriented case, and expressed respectively with "**,=**", "**-=**" and "**<=**". So all the properties valid for the set-oriented operators, can be easily extended to the instance-oriented case.

We now show how *ots* functions are related to *ts* functions to be evaluated inside set-oriented expressions, and which properties can be proved.

ots to ts	$ts(\mathcal{A}\text{+=}\mathcal{B},t) = min\{ots(\mathcal{A}\text{+=}\mathcal{B},t,oid)\},\ \forall oid \in R$ $ts(\mathcal{A}\text{<=}\mathcal{B},t) = min\{ots(\mathcal{A}\text{<=}\mathcal{B},t,oid)\},\ \forall oid \in R$ $ts(\mathcal{A}\text{,=}\mathcal{B},t) = min\{ots(\mathcal{A}\text{+=}\mathcal{B},t,oid)\},\ \forall oid \in R$ $ts(\text{-=}\mathcal{A},t) = max\{ots(\text{-=}\mathcal{A},t,oid)\},\ \forall oid \in R$
properties	$ots(\mathcal{A},t,oid) \leq ts(\mathcal{A},t)\ \forall oid$ $ts(\mathcal{A}\text{+=}\mathcal{B},t) \leq ts(\mathcal{A}\text{+}\mathcal{B},t)$ $\quad ts(\mathcal{A}\text{,=}\mathcal{B},t) \leq ts(\mathcal{A}\text{,}\mathcal{B},t)$ $ts(\mathcal{A}\text{<=}\mathcal{B},t) \leq ts(\mathcal{A}\text{<}\mathcal{B},t)$ $\quad ts(\text{-=}\mathcal{A},t) \geq ts(\text{-}\mathcal{A},t)$

4.4 Specification of rule triggering

The formal specification of rule triggering at time t for a rule r is given by the predicate $T(r,t)$: if the result of its evaluation is *true*, then the rule is triggered.

$$T(r,t) \stackrel{\text{def}}{=} R = \{e | e \in EB \wedge r.t_0 < timestamp(e) \leq t\} \wedge R \neq \emptyset \wedge\ \exists\, t'(r.t_0 < t' \leq t \wedge ts(r.\mathcal{E},t') > 0)$$

where $r.t_0$ is the time-stamp of the last consideration of the rule, while $r.\mathcal{E}$ is the triggering event expression of the rule. Observe that the predicate defines the set R which the *ts* function must be applied to as the set of all event occurrences more recent than the last consideration of the rule: in fact, the event calculus can be applied to a generic set of event occurrences; orthogonally, the triggering semantics defines this set.

Note that intuitively this semantics implies that a rule can be triggered only if something happened, otherwise the triggering mechanism ends because there is nothing which rules can react to. The reason of this choice ($R \neq \emptyset$) is that removing this constraint, a rule triggered by negated event types would always be fired even in absence of new event occurrences; then the system would become active instead of being reactive.

5 Implementation

The introduction of the event calculus language does not change the general architecture of the implementation of Chimera, described in [3], but affects only some specialized component, like the *Event Handler* and the *Trigger Support*: the former deals with event occurrences and stores them into the *Occurred Events* data structure; the latter maintains the current status of active rules (called *triggers* and chooses the trigger to be executed among those activated.

Chimera has a component, called *Block Executor*, which executes non interruptable execution blocks (user transaction lines or rule actions), finishes the execution of a block, it sends all the last generated event occurrences to the *Event Handler* in order to store them into the *Occurred Events* data structure. This data structure is maintaned as an event tree whose leaves are lists of event occurrences of the same type; furthermore, each leaves keeps the time-stamp of the more recent occurrence of the associated event type.

At this moment, the *Event Handler* calls the *Trigger Support* whose task is the determination of new activated rules. The *Trigger Support* maintains in the *Rule Table* the current status of all defined rules; this table is managed by means of a hash table, for fast access, but rules are also linked together by means of a queue on the basis of the priority order. To deal with composite events, each rule has two time-stamps associated to it: one, called *last-consideration*, stores the last consideration time-stamp; the other, called *last-consumption*, stores the time-stamp of the last event consumption, which is either the last consideration time if the rule is *consuming* or the initial time-stamp of the transaction if the rule is *preserving*. Another flag associated to a rule is the *triggered* flag, set to *true* if the rule is triggered or to *false* otherwise.

The *Trigger Support* checks for activated rules in the following way. It looks up into the *Rule Table* for all rules which are not triggered. When it finds one, it computes the ts value for the associated triggering event expression: if the computed value is positive, the rule is then triggered and the *triggered* flag is set to *true* (the rule will be detriggered once after its consideration).

Once new triggered rules are determined, the one to be executed is chosen by means of the rule queue, and passed to the *Block Executor*.

The evaluation of ts should take into account a certain number of things. At first, to determine the ts of a primitive event type is sufficient to query the *occurred events* table to get the last occurrence time-stamp of the desired event type E: if this time-stamp is not less than the value of *last-consideration*, this is the value of $ts(E,t)$, otherwise $ts(E,t)$ value is $-t$ (where t is the current time-stamp). Second, when dealing with instance-oriented operators, it is necessary to keep trace of all monitored events occurred on a single object: to do that, a sparse data structure can be associated to each rule and maintained until the consideration, then it is made empty; each item in this data structure stores the OID of an object affected by some event type since the last consideration and the list of event occurrences affecting that object since the last consideration.

5.1 Static Optimization

In general, the computation of the ts function for a given rule is an expensive task, especially if a large rule set has been defined. Our approach is to reduce the ts recomputation, by doing it only when it is highly probable that ts value becomes positive. The goal of the static optimization is to extract conditions on an event expression that guarantees, if not met, that the value of ts cannot become positive (recall rule triggering condition). This analysis should be performed when a rule is defined, and its results used to drive the *Trigger Support* in determining triggered rules.

The occurrence of composite event type $\mathcal{E}$, at time t, is indicated by the fact that the associated function ts assumes a new positive value at time t; thus, we need to check positive variations of ts, that we indicate as $\Delta^+(\mathcal{E})$. Depending on the composition operator, it may depend on positive or negative (indicated with $\Delta^-(\mathcal{E})$) variations of the component event expressions: the first case arises with conjunction, disjunction and precedence, the second one with negation.

	$\Delta^+(\texttt{-=}\mathcal{E}) \rightarrow \Delta_O^-(\mathcal{E})$
	$\Delta^-(\texttt{-=}\mathcal{E}) \rightarrow \Delta_O^+(\mathcal{E})$
	$\Delta_O^+(\texttt{-=}\mathcal{E}) \rightarrow \Delta_O^-(\mathcal{E})$
$\Delta^+(\texttt{-}\mathcal{E}) \rightarrow \Delta^-(\mathcal{E})$	$\Delta_O^-(\texttt{-=}\mathcal{E}) \rightarrow \Delta_O^+(\mathcal{E})$
$\Delta^-(\texttt{-}\mathcal{E}) \rightarrow \Delta^+(\mathcal{E})$	$\Delta^+(\mathcal{E}_1\texttt{<=}\mathcal{E}_2) \rightarrow \Delta_O^+(\mathcal{E}_2)$
$\Delta^+(\mathcal{E}_1\texttt{<}\mathcal{E}_2) \rightarrow \Delta^+(\mathcal{E}_2)$	$\Delta_O^+(\mathcal{E}_1\texttt{<=}\mathcal{E}_2) \rightarrow \Delta_O^+(\mathcal{E}_2)$
$\Delta^-(\mathcal{E}_1\texttt{-}\mathcal{E}_2) \rightarrow \Delta^-(\mathcal{E}_2)$	$\Delta^-(\mathcal{E}_1\texttt{<=}\mathcal{E}_2) \rightarrow \Delta_O^-(\mathcal{E}_2)$
$\Delta^+(\mathcal{E}_1\texttt{bin-op}\mathcal{E}_2) \rightarrow \Delta^+(\mathcal{E}_1), \Delta^+(\mathcal{E}_2)$	$\Delta_O^-(\mathcal{E}_1\texttt{<=}\mathcal{E}_2) \rightarrow \Delta_O^-(\mathcal{E}_2)$
$\Delta^-(\mathcal{E}_1\texttt{bin-op}\mathcal{E}_2) \rightarrow \Delta^-(\mathcal{E}_1), \Delta^-(\mathcal{E}_2)$	$\Delta^+(\mathcal{E}_1\texttt{bin-op=}\mathcal{E}_2) \rightarrow \Delta_O^+(\mathcal{E}_1), \Delta_O^+(\mathcal{E}_2)$
	$\Delta_O^+(\mathcal{E}_1\texttt{bin-op=}\mathcal{E}_2) \rightarrow \Delta_O^+(\mathcal{E}_1), \Delta_O^+(\mathcal{E}_2)$
	$\Delta^-(\mathcal{E}_1\texttt{bin-op=}\mathcal{E}_2) \rightarrow \Delta_O^-(\mathcal{E}_1), \Delta_O^-(\mathcal{E}_2)$
	$\Delta_O^-(\mathcal{E}_1\texttt{bin-op=}\mathcal{E}_2) \rightarrow \Delta_O^-(\mathcal{E}_1), \Delta_O^-(\mathcal{E}_2)$

Fig. 6. Derivation Rules.

$\{\Delta_O^+(\mathcal{E}), \Delta_O^-(\mathcal{E})\} \rightarrow \{\Delta_O(\mathcal{E})\}$	$\{\Delta^-(\mathcal{E}), \Delta_O^+(\mathcal{E})\} \rightarrow \{\Delta(\mathcal{E})\}$
$\{\Delta_O(\mathcal{E}), \Delta_O^+(\mathcal{E})\} \rightarrow \{\Delta_O(\mathcal{E})\}$	$\{\Delta_O(\mathcal{E}), \Delta^+(\mathcal{E})\} \rightarrow \{\Delta(\mathcal{E})\}$
$\{\Delta_O(\mathcal{E}), \Delta_O^-(\mathcal{E})\} \rightarrow \{\Delta_O(\mathcal{E})\}$	$\{\Delta_O(\mathcal{E}), \Delta^-(\mathcal{E})\} \rightarrow \{\Delta(\mathcal{E})\}$
$\{\Delta^+(\mathcal{E}), \Delta_O^+(\mathcal{E})\} \rightarrow \{\Delta^+(\mathcal{E})\}$	$\{\Delta^+(\mathcal{E}), \Delta^-(\mathcal{E})\} \rightarrow \{\Delta(\mathcal{E})\}$
$\{\Delta^-(\mathcal{E}), \Delta_O^-(\mathcal{E})\} \rightarrow \{\Delta^-(\mathcal{E})\}$	$\{\Delta^+(\mathcal{E}), \Delta(\mathcal{E})\} \rightarrow \{\Delta(\mathcal{E})\}$
$\{\Delta^+(\mathcal{E}), \Delta_O^-(\mathcal{E})\} \rightarrow \{\Delta(\mathcal{E})\}$	$\{\Delta^-(\mathcal{E}), \Delta(\mathcal{E})\} \rightarrow \{\Delta(\mathcal{E})\}$

Fig. 7. Simplification Rules.

This process can be performed until primitive event types are reached using a proper set of derivation rules (see Figure 6). In these rules for simplicity, we have used the symbol "bin-op" to indicate either the conjunction or the disjunction operator. These rules consider the instance-oriented operators as well; in order to deal with them, they use the symbols $\Delta_O^+(\mathcal{E})$, $\Delta_O^-(\mathcal{E})$ and $\Delta_O(\mathcal{E})$, which are analougous to the previous ones, but indicating *ots* variations for a single object. In the end, it leads to a set $V(\mathcal{E})$ of variations (positive or negative) for primitive event types describing whether or not the value of *ts* must be recomputed, because it might have changed, when new event occurrences arise; in practice, the conditions described by $V(\mathcal{E})$ are sufficient conditions ensuring that if new arising event occurrences do not match $V(\mathcal{E})$, no recomputation of *ts* is required.

Set $V(\mathcal{E})$ can be simplified using rules in Figure 7; in particular, with the symbol $\Delta(\mathcal{E})$ both a positive and negative variation is indicated. As an example, consider the following event expression $\mathcal{E}$.

$$\mathcal{E} \equiv ((A\texttt{+}B)\texttt{,}(C\texttt{+(-}A\texttt{))+((}A\texttt{+=}C\texttt{)<=(-=(}B\texttt{(<=}A\texttt{))}$$

The $V(\mathcal{E})$ set is obtained applying at first the derivation rules, then the simplification rules, as shown below.

$$
\begin{aligned}
V(\mathcal{E}) = \Delta^+(\mathcal{E}) \quad &= \\
&= \{\Delta^+((A\texttt{+}B)\texttt{,}(C\texttt{+}(\texttt{-}A))), \Delta^+((A\texttt{+=}C)\texttt{<=}(\texttt{-=}(B(\texttt{<=}A))\} \quad = \\
&= \{\Delta^+(A), \Delta^+(B), \Delta^+(C), \Delta^-(A), \Delta^+_O(A\texttt{+=}C), \Delta^-_O(B(\texttt{<=}A)\} \quad = \\
&= \{\Delta^+(A), \Delta^+(B), \Delta^+(C), \Delta^-(A), \Delta^+_O(A), \Delta^+_O(C), \Delta^-_O(B), \Delta^-_O(A)\} \quad = \\
&= \{\Delta(A), \Delta(B), \Delta^+(C)\}
\end{aligned}
$$

6 Conclusions

This paper has proposed an extension of event calculus for Chimera, characterized by the following features:

- It requires a minimal set of orthogonal operators.
- It continuously evolves the semantics of Chimera by enabling more sophisticated rule triggering, while preserving the other semantic features of the rule system.
- It supports a formal and efficient evaluation of triggering caused by event expressions of arbitrary complexity, based on the use of a function *ts* which associates each event expression to an integer value; a rule is triggered when the corresponding *ts* expression is positive, and not triggered otherwise.
- The function *ts* is assigned in such a way that certain obvious properties of calculus hold, such as De Morgan's rules or distributivity, associativity, and factoring of precedence expressions. Although this requirement seems mandatory to us, indeed it is not explicitly demontrated by some other event calculus proposals in the literature; achieving this result has required to us a nonobvious "twisting" of the *ts* functions.
- As an optimization, the evaluation of the *ts* function is required when certain operations occur which have the potential of "changing the sign" of *ts*, and can be skipped otherwise.

Given the above features, we believe that the proposed event calculus applies not only to Chimera, but also to all other systems which currently support individual or disjunctive events (including all relational products which support triggers).

References

1. H. Branding, A. Buchmann, T. Kudrass, and J. Zimmermann. Rules in an open system: The reach rule system. In *Proc. of the 1st Int. Workshop on Rules in Database Systems*, pages 127–142, Edimburgh, August 1993.
2. H. Branding, A. Buchmann, T. Kudrass, and J. Zimmermann. Rules in an open system: The reach rule system. In *Proc. of the 1st Int. Workshop on Rules in Database Systems*, pages 111–125, Edimburgh, August 1993.
3. S. Castangia, G. Guerrini, D. Montesi, and G. Rodriguez. Design and implementation for the active rule language of chimera. In *DEXA-95 6th international Workshop and Conference on Database and Expert Systems Applications*, London, UK, September 1995.

4. S. Ceri, P. Fraternali, S. Paraboschi, and L. Tanca. Active rule management in chimera. In *[23]*.
5. S. Ceri and R. Manthey. Consolidated specification of chimera. Technical Report IDEA DE.2P.006.01, November 1993.
6. S. Chakravarthy, E. Anwar, L Maugis, and D. Mishra. Design of sentinel: an object-oriented dbms with event-based rules. *Information and Software Technology*, 36(9), 1994.
7. S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S. K. Kim. Composite events for active databases: Semantics, context and detection. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 606–617, Santiago, Chile, September 1994.
8. U. Dayal, A. P. Buchmann, and S. Chakravarthy. The hipac project. In *[23]*.
9. U. Dayal, A. P. Buchmann, and D. R. McCarthy. Rules are objects too: A knowledge model for an active object-oriented database system. In K. R. Dittrich, editor, *Proceedings of the 2nd International Workshop on Object-Oriented Databases*. Springer-Verlag, 1988. LNCS 334.
10. P. Fraternali and L. Tanca. A structured approach for the definition of the semantics of active databases. June 1995. submitted to ACM-TODS.
11. S. Gatziu and K. R. Dittrich. Events in an active object-oriented database system. In *Proc. of the 1st Int. Workshop on Rules in Database Systems*, pages 23–39, Edimburgh, August 1993.
12. N. H. Gehani and H. V. Jagadish. Ode as an active database: Constraints and triggers. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 327–336, Barcelona, Spain, September 1991.
13. N. H. Gehani, H. V. Jagadish, and O. Shmueli. Composite event specification in active databases: Model and implementation. In *Proceedings of the 18th International Conference on Very Large Data Bases*, pages 327–338, Vancouver, Canada, 1992. British Columbia.
14. N. H. Gehani, H. V. Jagadish, and O. Shmueli. Event specification in an active object-oriented database. In *1992 ACM SIGMOD*, pages 81–90, San Diego, CA, USA, May 1992.
15. E. N. Hanson. Rule condition testing and action execution in ariel. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 327–336, Barcelona, Spain, September 1991.
16. E. N. Hanson, M. Chaabouni, C-H. Kim, and Y-W. Wang. A predicate matching algorithm for database rule systems. *ACM Journal*, pages 271–280, May 199O.
17. ISO-OSI. *SQL3 Document X3H2-94-080 and SOU-003, ISO-ANSI Working Draft*, 1994.
18. R. Maiocchi and B. Pernici. Temporal data management systems: A comparative view. *IEEE Transactions on Knowledge and Data Engineerging*, 3(4):504–524, December 1991.
19. W. Naqvi and M. T. Ibrahim. Rule and knowledge management in an active database system. In *[23]*.
20. N. W. Paton, O. Diaz, M. H. Williams, J. Campin, A. Dinn, and A. Jaime. Dimensions of active behaviour. In *Proc. of the 1st Int. Workshop on Rules in Database Systems*, pages 40–57, Edimburgh, August 1993.
21. M. Stonebraker, A. Jhingran, J. Goh, and S. Potamios. On rules, procedures, chaching, and views in data base systems. In *Proc.ACM-SIGMOD Int. Conference*, pages 281–290, Atlantic City, June 1990.

22. M. Teisseire, P. Poncelet, and R. Cicchetti. Towards event-driven modelling for database design. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 327–336, Santiago, Chile, September 1994.
23. J. Widom and S. Ceri. *Active Database Systems.* Morgan Kaufmann, San Matteo, California, August 1995.
24. J. Widom, R. J. Cohrane, and B. G. Lindsay. Implementing set-oriented production rules as an extension od starburst. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 275–285, Barcelona, Spain, September 1991.
25. J. Widom and S. J. Finkelstein. Set-oriented production rules in relational database systems. In *Proc.ACM-SIGMOD Int. Conference*, pages 250–270, Atlantic City, June 1990.