Composite Temporal Events in Active Databases: A Formal Semantics

Iakovos Motakis
Department of Computer Science, U.C.L.A.
Los Angeles, California

Carlo Zaniolo
Department of Computer Science, U.C.L.A.
Los Angeles, California

Abstract

Active databases must support rules triggered by complex patterns of composite temporal events. This paper proposes a general method for specifying the semantics of composite event specification languages. The method is based on a syntax-directed translation of the composite event expressions into $Datalog_{1S}$, whose formal semantics is then used to define the meaning of the original event expressions. We show that the method is applicable to languages such as ODE, Snoop and SAMOS that are based respectively on the formalisms of Finite State Machines, Event Graphs and Petri Nets. The proposed method overcomes various problems and limitations affecting such formalisms.

1 Introduction

A new generation of database systems supports active rules for the detection of events occurring in the database and the triggering of induced actions. In many applications, the simple-event detection mechanisms found in commercial systems, such as Ingres, Oracle or Sybase, are not sufficient; complex sequences of events must instead be detected as the natural precondition for taking actions and firing rules [9]. Sophisticated research prototypes have in fact been implemented recently to provide this capability— an incomplete list of such systems includes [9, 13, 10, 4]. These systems feature composite event detection mechanisms, which are based on formalisms such as Finite State Automata [12], Petri Nets [11], or Event Graphs [3].

The problem of formally specifying the semantics of active rules remains largely unsolved. Indeed, giving a formal semantics to active database behavior presents a challenge even when only simple events are involved. For composite event expressions, this problem becomes even more complex, inasmuch as the issues of temporal databases and active rule languages are now combined and intertwined [17]. This issue must be resolved if trustworthyness is expected from systems incorporating this advanced capability.

In this paper we propose a general method for defining the semantics of composite event specification languages. The method is based on a syntax-directed translation of the composite event expressions into $Datalog_{1S}$, whose formal semantics is then used to define the meaning of the original event expressions.

We first demonstrate the method by giving a complete definition of the Event Pattern Language (EPL). EPL, the language of an active database system designed and implemented at UCLA provides the capability of detecting, reasoning and acting upon complex patterns of events that evolve over time. Then, we explain how our method can be applied to ODE [13], Snoop [4] and SAMOS [10], in order to formally define the intuitive and/or operational semantics of the composite event specification languages of these systems. Furthermore, we point out their differences along with some problems and limitations of theirs, which often follow from their implementation frameworks.

The method proposed in this paper matches and surpasses the capabilities of the formalisms used in the past, in terms of (a) defining precisely the intuitive semantics of different language characteristics, including *event attributes, negation, simultaneous events and parameter contexts*, and (b) ease of mapping to correct execution semantics. Therefore, we obtain a general method for defining the semantics of such languages, and a useful tool for comparing their constructs and their expressive power.

Because of space limitations, explicit time events are not discussed in this paper. However, these can also be incorporated in our method using the approach discussed in [15].

2 The EPL Language

In the rule-based syntax of EPL, sequences of goals specify sequences of events; each goal can correspond to either (i) a basic event, or (ii) a (possibly negated) basic event qualified by condition predicates, or (iii) a composite event constructed using sequences, repetitions, conjunctions, disjunctions and negations of other (possibly composite) events. EPL is implemented as a portable front-end to active relational databases supporting simple event detection. Two versions of EPL have been developed, the first in $\mathcal{CDL}$++ [1] and the second in CLIPS [16].

2.1 EPL Programs

An EPL program is organized in *modules*, which can be compiled and enabled independently. The *events declaration section* of a module defines the set of relevant basic event types, monitored by the module. A *basic event type* is denoted as :

```
insert(Rname), delete(Rname), update(Rname),
```

where `Rname` is the name of a database relation.

EPL rules are specified in a module's *rules section*. Each rule has a name, which is unique within its module. A rule's body (head) corresponds to an event (action). Example 1 demonstrates an EPL module with one rule. In all examples, we use the following bank accounts relation.

```
ACC(Accno, Owner, Type, Balance)
```

Example 1 *An EPL module with a rule that keeps track of large withdrawals from savings accounts.*

```
begin AccMonitor
   monitor  insert(ACC), update(ACC), delete(ACC);
   LargeWithdrawal:
      update( ACC(X),
              X.Type = "Savings", X.old_Balance-X.new_Balance > 100000 )
   -> write( "Large withdrawal from account %d at time %s \n",
              X.Accno, asctime( X.evtime) ).
end.
```

The **AccMonitor** module keeps track of all the modifications in the **ACC** relation. Rule **LargeWithdrawal** specifies a *qualified basic event*. Such an event has the form:

$$\texttt{evtkind(}\ \texttt{Rname(X)},\ <\ \texttt{condition} - \texttt{expression}\ >\ \texttt{)},$$

where **X** denotes the tuple of relation **Rname**, that has been inserted, deleted or updated. For *update* events, EPL makes available to the programmer both the new and the old contents of the updated tuple. The prefixes **new** and **old** are used to distinguish between them. When no prefix is specified, the new contents are assumed.

The attribute **evtime**, which is attached to a basic event's tuple variable contains the time the event occurred. The < condition-expression > is built using the standard arithmetic and comparison operators and the logical connectives AND, OR and NOT.

Actions: There are three kinds of actions : (a) Write actions, (b) SQL commands and (c) Operating system calls.

In all cases, the format of the action specifier is similar to that of the *printf* statement in the C language. The action's arguments are taken from tuple variables defined by basic events in the rule's body.

Negated Events: A (qualified) basic event may be negated. Consider for instance the following rule, as part of the module **AccMonitor**:

```
NoUpdateOn00201:
      ! update( ACC(X), X.Accno = 00201 )
      -> write("any event but an update on 00201")
```

This *negated qualified basic event* will be satisfied by the occurrence of *any* basic event of the types monitored by the **AccMonitor** (including possible insertions and deletions), except for an update of account 00201.

2.2 EPL Language Constructs

So far, we have been concerned with *basic events*, which may be qualified or negated.

The power of EPL follows from its ability to specify composite events. We distinguish between *event expressions* (also called *event types*) and *event instances*. An event expression **E** is specified using the EPL language, where an event instance of **E** consists of a sequence of basic events that participated in the satisfaction of **E**. In the sequel, we will refer to *events* when the distinction

is clear from the context. We will also use the term *event occurrence*, to refer to the time instant when an event expression is satisfied (an instance of this event expression is completed).

Composite event expressions are defined as follows:

Definition 1 *Let $E_1, E_2, ..., E_n$, $n > 1$, be EPL event expressions (maybe composite themselves). The following is also an EPL event expression :*

1. $(E_1, E_2, ..., E_n)$ *: a sequence consisting of an instance of* E_1, immediately *followed by an instance of* E_2, ..., immediately *followed by an instance of* E_n.
2. $*:E$ *: a sequence of zero or more consecutive instances of* E.
3. $(E_1 \ \& \ E_2 \ \& \ ... \ \& \ E_n)$ *: A conjunction of events. It occurs when all events* $E_1, ..., E_n$ *occur simultaneously.*
4. $\{E_1, E_2, ..., E_n\}$ *: A disjunction of events. It occurs when at least one event among* $E_1, ..., E_n$ *occurs.*
5. $!E$ *: It occurs when not any instance of* E *occurs.*

A number of additional (derived) constructs may be defined in terms of the basic ones (see also [12]). Some of these are :

- $any \equiv$ *The disjunction of all basic events (of the types monitored). It occurs every time such an event occurs.*
- $[E_1, E_2, ..., E_n] \equiv (E_1, *:any, E_2, *:any, ..., *:any, E_n)$. *Relaxed sequence. It consists of an instance of* E_1, *followed* later *by an instance of* E_2, ..., *followed* later *by an instance of* E_n.
- $prior(E_1, E_2) = [E_1, any] \ \& \ E_2$. *An occurrence of* E_2 *follows an occurrence of* E_1 *(i.e., an instance of* E_1 *is completed prior to the completion of an instance of* E_2*)*
- $first(E) \equiv (E \ \& \ ![E, any])$*: It occurs, when the first instance of* E *occurs.*

Note that in an instance of $[\mathtt{E_1}, \mathtt{E_2}]$, the first basic event in the instance of $\mathtt{E_2}$ must follow an occurrence of $\mathtt{E_1}$, where in $\mathtt{prior(E_1, E_2)}$, this is not required.

In addition to the above, a composite event may have attributes, which are derived from the attributes of its component basic events. Attribute semantics and scope rules are described in the next section. Examples of EPL composite events follow.

Example 2 *Report transfers of large amounts, from a customer's savings account to his/her checking account.*

```
LargeTransfer:
   ( update(ACC(X),
            X.Type = "Savings", X.old_Balance-X.new_Balance > 100000),
     update(ACC(Y),
            Y.Type = "Checking", Y.Owner = X.Owner,
            Y.new_Balance-Y.old_Balance = X.old_Balance-X.new_Balance) )
   -> write("Large Transfer of %s \n", X.Owner)
```

We assumed here, that a transfer transaction results in an immediate sequence of updates. The condition expression of the second update can refer to the tuple variables of both basic events.

Example 3 *Report the cases where two large deposits are made to an account, without any intervening withdrawal from it.*

```
Good Customer:
   ( update(ACC(X), X.new_Balance-X.old_Balance > 100000),
     *:! update(ACC(Y),
                Y.Accno = X.Accno, Y.new_Balance < Y.old_Balance),
     update(ACC(Z),
            Z.Accno = X.Accno, Z.new_Balance-Z.old_Balance > 100000)
   -> write("Good Customer: %s \n", X.Owner)
```

Example 4 *(Relaxed Sequence) Identify cases of customers who opened a Savings account, after they had closed another one before.*

```
WellcomeBack:
   [ delete(Acc(X), X.Type = Savings),
     insert(Acc(Y), Y.Type = Savings, Y.Owner = X.Owner) ]
   -> write("Customer %s is back \n", X.Owner)
```

Quite often, we need to find the *first* instance of an event F following an instance of another event E. Below, we express such an event pattern using the definition of the `first` derivative construct. This demonstrates the negation of a composite event and a conjunction.

Example 5 *If an account's balance drops to zero, report the next deposit to it, as well as the time elapsed.*

```
ActiveAgain:
   [ update(ACC(X), X.Balance = 0) ,
     ( update(ACC(Y), Y.Accno = X.Accno, Y.Balance > 0)  &
       ! [update(ACC(Z), Z.Accno = X.Accno, Z.Balance > 0),  any] ) ]
  ->  write("Account %d is active again, after so much time: %s \n",
            X.Accno, asctime(Y.Time - X.Time) )
```

3 Semantics of EPL

We first introduce the notion of *event histories*, against which the EPL expressions are evaluated. The *global event history* is a series of *basic events*, that is ordered by time of occurrence (timestamp) and can be obtained from a system log. It can be represented by the relation hist(EventType, RelName, TimeStamp), where each tuple records a basic event occurrence and contains its type (insert, delete, or update), the name of the relation upon which it occurred, and the time of the occurrence.

Since an EPL expression is evaluated with respect to a particular module, a separate *event history* must be obtained for each such module. Focusing now on one module, we assume that a relation evt_monit(EventType, TableName) is kept for it, that records the basic event types the module monitors. Then, our module's *event history* is defined by the following stratified *Datalog* rules:

```
hist_monit(nil, nil, 0000, 0.)
hist_monit(E, R, T2, s(J)) ←   hist_monit(_, _, T1, J),
                               hist(E, R, T2), evt_monit(E, R),
                               ¬between(T1, T2).
between(T1, T2) ←   hist(E, R, T),  evt_monit(E, R),  T1 < T,  T < T2.
```

In this way, an *event history* can be defined for each module of interest. For nstance, the following table contains a brief example event history for module `AccMonitor`:

hist_monit	EventType	TableName	TimeStamp	Stage
	nil	nil	0000	0
	upd	ACC	1423	1
	upd	ACC	1425	2
	ins	ACC	1430	3
	ins	ACC	1502	4

Observe that a sequence number, called *stage* has been introduced. The stage sequence defines an ordered structure on the distinct timestamps, that allows us to express properties of composite events that are based on the *relative order* of occurrence of their component basic events, as opposed to absolute time properties. Thus, the stage is the unit of time (*chronon*) in our model. Absolute time properties of events can also be expressed using their timestamps.

Different *event occurrence granularities* can be handled. At the "smallest database operation" granularity, every new insertion, deletion, or update creates a new stage. However, if "transaction boundaries granularity" is assumed, then each committed transaction creates a new stage, and all the basic events that occurred within this transaction are recorded in its stage, timestamped with the transaction's commit time. Basic events that share a sequence (stage) number are called *simultaneous* and are further discussed in Section 3.5.

Using this model, the fundamental concept of an *immediate sequence* of basic events corresponds to the occurrence of such events at successive stages.

These observations lead naturally to the use of $Datalog_{1S}$ as a formal basis for defining the semantics of EPL rules. $Datalog_{1S}$ [2] is a temporal language that extends *Datalog*, by allowing every predicate to have at most one temporal argument (constructed using the unary successor function s), in addition to the usual data arguments. The temporal argument in our case is the *stage* column in the event history.

As most active relational databases do, we further assume that for each DB table, there are three relations accumulating the inserted, deleted and updated tuples, together with their timestamps. For inserts into ACC for instance, we have the relation `ins_ACC(Accno, Owner, Type, Balance, Timestamp)`. The `del_ACC` table has a similar format, while for updates, we must record both the old and new values:

`upd_ACC(Accno_old, Accno_new, Owner_old, Owner_new, ... Timestamp)`

3.1 Event Satisfaction

We can now define the meaning of arbitrary EPL event expressions, through the notion of *satisfaction* of such expressions.

We start with qualified basic events. For instance, the satisfaction of the event `E = ins(ACC(X), X.Type = "Savings")` is defined as follows:

$$\texttt{ins\_ACC}(\texttt{Accno}, \texttt{Owner}, \texttt{Type}, \texttt{Bal}, \texttt{Time}, \texttt{J}) \leftarrow \texttt{hist\_monit}(\texttt{ins}, \text{“ACC”}, \texttt{Time}, \texttt{J}), \texttt{insertedACC}(\texttt{Accno}, \texttt{Owner}, \texttt{Type}, \texttt{Bal}, \texttt{Time}).$$

$$\texttt{sat}_{\texttt{E}}(\texttt{Accno}, \texttt{Owner}, \texttt{Type}, \texttt{Bal}, \texttt{Time}, \texttt{J}) \leftarrow \texttt{ins\_ACC}(\texttt{Accno}, \texttt{Owner}, \texttt{Type}, \texttt{Bal}, \texttt{Time}, \texttt{J}), \quad \texttt{Type} = \text{“Savings”}.$$

The predicate ins_ACC describes the history of occurrences of insertedACC; for each occurrence of this event type, ins_ACC contains a tuple with its attribute bindings and the stage of the occurrence.

In general, a qualified basic event is represented as $\mathtt{E} = \mathtt{evtkind}(\mathtt{R}(\mathtt{X}),\ \mathtt{q}(\mathtt{X}))$, where q denotes the event's condition expression, which can refer to the attribute values of tuple variable X. The rule template for the *satisfaction predicate* of such an event is:

$$\mathtt{sat_E}(\mathtt{X},\mathtt{J}) \leftarrow\ \ \mathtt{evtkind\_R}(\mathtt{X},\mathtt{J}),\ \ \mathtt{q}(\mathtt{X})$$

The concept of "an event *immediately following* another event" can also be expressed. Take for instance, the immediate sequence of Example 2, which is represented as:

$$\mathtt{F} = (\ \mathtt{upd}(\mathtt{ACC}(\mathtt{X}), \mathtt{q_1}(\mathtt{X})),\ \ \mathtt{upd}(\mathtt{ACC}(\mathtt{Y}), \mathtt{q_2}(\mathtt{X},\mathtt{Y}))\ )$$

Its semantics is defined by the following three $Datalog_{1S}$ rules (from now on, unless otherwise indicated, variables will denote tuples):

Example 6

$$\begin{array}{ll} \mathtt{sat_1}(\mathtt{X},\mathtt{J}) \leftarrow & \mathtt{upd\_ACC}(\mathtt{X},\mathtt{J}),\ \ \mathtt{q_1}(\mathtt{X}). \\ \mathtt{sat_2}(\mathtt{X},\mathtt{Y},\mathtt{s}(\mathtt{J})) \leftarrow & \mathtt{upd\_ACC}(\mathtt{Y},\mathtt{s}(\mathtt{J})),\ \ \mathtt{sat_1}(\mathtt{X},\mathtt{J}),\ \ \mathtt{q_2}(\mathtt{X},\mathtt{Y}). \\ \mathtt{sat_F}(\mathtt{X},\mathtt{Y},\mathtt{J}) \leftarrow & \mathtt{sat_2}(\mathtt{X},\mathtt{Y},\mathtt{J}). \end{array}$$

The first qualified basic event occurs at stage J, if an update on relation ACC is recorded at this stage and condition $\mathtt{q_1}$ is satisfied. The second update on ACC must then occur at the next stage s(J) and condition $\mathtt{q_2}$ must be satisfied (observe that $\mathtt{q_2}$ can refer to the tuple variable X defined by the first basic event, in addition to Y). The third rule is a *copy-rule*, inasmuch as the satisfaction of composite event F coincides with that of $\mathtt{sat_2}$.

There exists a natural mapping from EPL expressions to $Datalog_{1S}$. Thus, to formally define the meaning of an EPL expression, we only need to define a procedure which derives an equivalent set of $Datalog_{1S}$ rules for that expression. The resulting set of rules has a well-established formal semantics (model-theoretic and fixpoint-based) [2]. To formalize the translation, we represent EPL expressions by their parse trees, using the following prefix notation:

1. $seq(E_i, E_j) \equiv (E_i, E_j).$[1]
2. $*seq(E_i, E_j) \equiv (*E_i, E_j).$[2]
3. $and(E_i, E_j) \equiv E_i\ \&\ E_j.$
4. $or(E_i, E_j) \equiv \{E_i, E_j\}.$
5. $neg(E) \equiv !E.$

Example 7 *The EPL expression*

$$(\mathtt{upd}(\mathtt{A}(\mathtt{X}),\mathtt{q_a}(\mathtt{X})),\ *:(*:\mathtt{ins}(\mathtt{B}(\mathtt{Y}),\mathtt{q_b}(\mathtt{X},\mathtt{Y})),\ \mathtt{del}(\mathtt{C}(\mathtt{Z}),\mathtt{q_c}(\mathtt{X},\mathtt{Z}))),\ \mathtt{upd}(\mathtt{D}(\mathtt{V}),\mathtt{q_d}(\mathtt{X},\mathtt{V}))$$

The parse tree for the expression of Example 7 is shown in Figure 1. The nodes of the tree are numbered according to the postorder traversal sequence.

Each node i corresponds to a subevent $\mathtt{E_i}$, and the *satisfaction predicate* of $\mathtt{E_i}$ is denoted as $\mathtt{sat_i}$. For a subevent expression, its *satisfaction predicate* contains one tuple for each distinct (in terms of variable bindings and stage) occurrence of this subevent.

[1] $(E_1, E_2, \ldots, E_{n-1}, E_n) = seq(E_1, seq(E_2, \ldots, seq(E_{n-1}, E_n)\ldots))$. Similarly for the relaxed sequence, the conjunction and the disjunction constructs.

[2] We use the binary construct *seq in place of the *: EPL construct, so that the representation is more compact and easier to follow. This is not restricting, since $*:\mathtt{E} \equiv *\mathtt{seq}(\mathtt{E}, \epsilon)$, where $\epsilon \equiv$ no event.

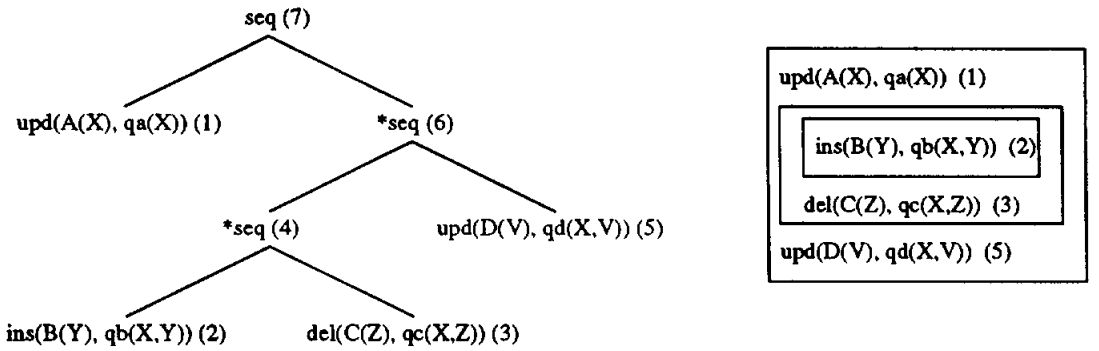

Figure 1: An EPL expression with Immediate and Star Sequences

3.2 The Translation Procedure

As demonstrated by the last translation example, for a composite EPL expression, the $Datalog_{1S}$ rules must model (i) the transmission of variable bindings according to the scope rules of the various constructs, so that variables can be matched and conditions can be checked, and (ii) the temporal precedences among the various subevents.

Table 1 describes how this information is derived for each basic EPL construct (formally it defines a simple attribute grammar for syntax-directed translation).

EvtType E	*PPS*	*EVar(E)*	*IVar*
evt(R(X))	—	X	—
seq(F,G)	$PPS(F) = PPS(E)$ $PPS(G) = F$	$EVar(F) \cup EVar(G)$	$IVar(F) = IVar(E)$ $IVar(G) = IVar(E) \cup EVar(F)$
**seq(F,G)*	$PPS(F) = F \cup PPS(E)$ $PPS(G) = F \cup PPS(E)$	$EVar(G)$	$IVar(F) = IVar(E)$ $IVar(G) = IVar(E)$
or(F,G)	$PPS(F) = PPS(E)$ $PPS(G) = PPS(E)$	$\emptyset$	$IVar(F) = IVar(E)$ $IVar(G) = IVar(E)$
and(F,G)	$PPS(F) = PPS(E)$ $PPS(G) = PPS(E)$	$EVar(F) \cup EVar(G)$	$IVar(F) = IVar(E)$ $IVar(G) = IVar(E)$
neg(F)	$PPS(F) = PPS(E)$	$\emptyset$	$IVar(F) = IVar(E)$

Table 1: An attribute grammar for syntax-directed translation from EPL to $Datalog_{1S}$.

For each subevent `Q` of an EPL event `E`, the second column in Table 1 defines the *Possible Predecessors Set* of `Q`, denoted as `PPS(Q)`. A subevent `P` is a possible predecessor of `Q` within `E`, if in an instance of `E`, the satisfaction of `P` can *immediately* precede the first basic event of an instance of `Q` (i.e., the instance of `Q` can begin at the next stage). Because of disjunctions and the star operator, a particular subevent may have many possible predecessors.

For example, consider the immediate sequence event: `E = seq(F,G)`. `F` is the only possible predecessor of `G`; but the set of possible predecessors of `F` depends on which events may precede `E`—i.e., `F` inherits `E`'s possible predecessors.

The remaining two columns of Table 1 describe the scope rules for variables in EPL. The third column shows the set of *exported variables* of an EPL expression. These are variables defined in the expression (variables appearing in basic events within this expression), whose scopes extends past the satisfaction of the expression. The fourth column contains for each subevent `Q` of an EPL expression, the set of variables *imported* into `Q` (variables defined outside `Q`, whose scopes extends to `Q`).

Again, for `E = seq(F,G)`, the set of variables exported from `E` is the union of the variables exported from `F` and `G`. On the other hand, `E` might have imported some variable names from previous events and if so, these are also passed down to `F` and `G`. In addition to variables inherited by `E`, variables imported into `G` include those exported from `F`.

Event Type E	*Datalog$_{1S}$ Rule Templates*
Qual. Basic Event evt(R(X), Cond)	*for each* $P \in PPS(E)$ $sat_E(IV, X, s(J)) \leftarrow evt\_R(X, s(J)),\ sat_P(IV, \_, J),$ $Cond(IV, X)$
seq(F(X),G(Y))	$sat_E(IV, X, Y, J) \leftarrow sat_G(IV, X, Y, J)$
**seq(F(X),G(Y))*	$sat_E(IV, Y, J) \leftarrow sat_G(IV, Y, J)$
or(F(X),G(Y))	$sat_E(IV, J) \leftarrow sat_F(IV, X, J)$ $sat_E(IV, J) \leftarrow sat_G(IV, Y, J)$
and(F(X),G(Y))	$sat_E(X, Y, IV, J) \leftarrow sat_F(IV, X, J),\ sat_G(IV, Y, J)$
neg(F(X))	*for each* $P \in PPS(E)$ $sat_E(IV, s(J)) \leftarrow any(s(J)),\ sat_P(IV, \_, J),$ $\neg sat_F(\_, s(J))$
ε	*for each* $P \in PPS(E)$, $sat_E(IV, J) \leftarrow sat_P(IV, \_, J)$

Table 2: *Datalog$_{1S}$* rule templates for the basic constructs of EPL.

Using the information of Table 1, the generation of the actual rules is simple as shown in Table 2. Observe that except for basic events, `X` and `Y` denote *sets* of exported variables defined in various subevents, where `IV` denotes the *set* of imported variables into a particular event type `E`. The anonymous variable _ has replaced all variables that must be kept local.

The first row of this table deals with qualified basic events having some possible predecessors (the case of a basic event with no possible predecessors is trivial). Such an event `E` is satisfied at stage `s(J)`, when: (1) a possible predecessor of E was satisfied at stage `J`, (2) `E` occurs at stage `s(J)`, and (3) The condition of `E` is satisfied. Example 6 illustrates this translation.

The rules for disjunction and conjunction are apparent. Observe that in a conjunction, all the variables defined in its conjuncts are exported, where in a disjunction, none of the variables defined in its disjuncts is exported. The rule for *negated events* is explained in Section 3.6.

Note also, that the variables of a satisfaction predicate consists of the union of its exported variables, plus the variables imported into it.

3.3 Immediate and Star Sequences

Having illustrated how immediate sequences are handled, we move on to the case of star sequences, which is somewhat more complicated.

Consider e.g., the EPL expression E = (F, G, *:H, K). Obviously, PPS(G) = {F}. However, because of the star operator, an instance of H might immediately follow either an occurrence of G, or a previous occurrence of H. Therefore, PPS(H) = {G,H}. Similarly, an instance of K may immediately follow either an occurrence of G (zero instances of H after G), or the last occurrence of H and thus, PPS(K) = {G,H}. Variables defined in a *star* subexpression are not exported to subexpressions that follow. The fourth row of Table 1 provides the formal details.

Example 7 shows a more complicated case, where *star* subexpressions are nested. Referring to Figure 1 and using Table 1, we get:

$$
\begin{aligned}
PPS(7) &= \emptyset \\
PPS(1) &= PPS(7) = \emptyset \\
PPS(6) &= \{1\} \\
PPS(4) &= \{4\} \cup PPS(6) = \{1,4\} \\
PPS(5) &= \{4\} \cup PPS(6) = \{1,4\} \\
PPS(2) &= \{2\} \cup PPS(4) = \{1,2,4\} \\
PPS(3) &= \{2\} \cup PPS(4) = \{1,2,4\}
\end{aligned}
$$

The variable scopes for this example have been visualized in Figure 1 using contours. Basic events are listed in order of their appearance in the EPL expression and all basic events in the same *star* subexpression are enclosed within the same contour. The condition of a basic event E can refer to all variables whose scopes extends to this event. Using this information and the *PPS* sets of the basic events, the following $Datalog_{1S}$ rules are derived for Example 7:

$$
\begin{aligned}
sat_1(X,J) &\leftarrow upd\_A(X,J),\ q_a(X). \\
sat_2(X,Y,s(J)) &\leftarrow ins\_B(Y,s(J)),\ sat_1(X,J),\ q_b(X,Y). \\
sat_2(X,Y,s(J)) &\leftarrow ins\_B(Y,s(J)),\ sat_2(X,\_,J),\ q_b(X,Y). \\
sat_2(X,Y,s(J)) &\leftarrow ins\_B(Y,s(J)),\ sat_4(X,\_,J),\ q_b(X,Y). \\
sat_3(X,Z,s(J)) &\leftarrow del\_C(Z,s(J)),\ sat_1(X,J),\ q_c(X,Z). \\
sat_3(X,Z,s(J)) &\leftarrow del\_C(Z,s(J)),\ sat_2(X,\_,J),\ q_c(X,Z). \\
sat_3(X,Z,s(J)) &\leftarrow del\_C(Z,s(J)),\ sat_4(X,\_,J),\ q_c(X,Z). \\
sat_4(X,Z,J) &\leftarrow sat_3(X,Z,J). \\
sat_5(X,V,s(J)) &\leftarrow upd\_D(V,s(J)),\ sat_1(X,J),\ q_d(X,V). \\
sat_5(X,V,s(J)) &\leftarrow upd\_D(V,s(J)),\ sat_4(X,\_,J),\ q_d(X,V). \\
sat_6(X,V,J) &\leftarrow sat_5(X,V,J). \\
sat_E(X,V,J) &\leftarrow sat_6(X,V,J).
\end{aligned}
$$

Consider for instance $E_2 = ins(B(Y), q_b(X,Y))$. This basic event may immediately follow an occurrence of basic event E_1, or another occurrence of E_2 (because of the innermost star), or an occurrence of a star subsequence E_4 (because of the outermost * iteration).

The satisfaction predicates for the seq and *seq nodes are defined through *copy-rules*. These predicates are not needed, unless such a node is a possible predecessor of some basic event, as is the case of E_4. However, we have included them in our presentation for clarity reasons.

As demonstrated in this example, EPL scope rules are implemented, by passing variables through the satisfaction predicates, to the conditions of all the basic events within the scope of the variables.

3.4 Any and Relaxed Sequences

In section 2.2, EPL derivative constructs such as **any**, **prior**, and **relaxed sequences** were defined in terms of the basic constructs. Thus, a translation into $Datalog_{1S}$ need not be given explicitly. Yet, a direct translation is often desirable, as it leads to much more efficient implementation. For instance, **any** need not be defined as the disjunction of all basic events in the module of interest, but can be simply derived as follows:

$$\texttt{any(J)} \leftarrow \texttt{hist\_monit}(\_, \_, \_, \texttt{J})$$

A relaxed sequence is treated similarly to an immediate sequence; e.g. the rules of Table 1 for an immediate sequence remain intact in the case of a relaxed sequence. The only difference is that in [F, G], an instance of G may start at some stage later, but not necessarily immediately after an occurrence of F. By using an *auxiliary predicate* $\texttt{has\_sat}_1$, the relaxed sequence

$$\texttt{E} = [\ \texttt{upd(ACC(X)}, \texttt{q}_1\texttt{(X))},\ \texttt{upd(ACC(Y)}, \texttt{q}_2\texttt{(X,Y))}\ ],$$

can be translated into the following rules:

$$\begin{array}{ll}
\texttt{sat}_1\texttt{(X,J)} \leftarrow & \texttt{upd_ACC(X,J)},\ \texttt{q}_1\texttt{(X)} \\
\texttt{has_sat}_1\texttt{(X,J)} \leftarrow & \texttt{sat}_1\texttt{(X,J)} \\
\texttt{has_sat}_1\texttt{(X,s(J))} \leftarrow & \texttt{any(s(J))},\ \texttt{has_sat}_1\texttt{(X,J)} \\
\texttt{sat}_2\texttt{(X,Y,s(J))} \leftarrow & \texttt{upd_ACC(Y,s(J))},\ \texttt{has_sat}_1\texttt{(X,J)},\ \texttt{q}_2\texttt{(X,Y)}
\end{array}$$

3.5 Conjunction and Simultaneous Events

A *conjunctive event* E = (F & G) occurs at a stage where both F and G occur. The instances of F and G that cause E to be satisfied may have different starting stages. F and G are *evaluated independently of each other* (in parallel).

Using the conjunction construct, we can express sequences based on event occurrences, as opposed to event instances that follow each other. An example is the definition of **prior**, which is repeated here (variables are included):

$$\texttt{E(X,Y)} = \texttt{prior(F(X),G(Y))} = (\ [\texttt{F(X)}, \texttt{any}]\ \&\ \texttt{G(Y)}\ )$$

Assuming that the rules for F(X) and G(Y) have been generated and that an auxiliary predicate $\texttt{has\_sat}_\texttt{F}$ is defined as in the previous section, the satisfaction predicate of E is defined as:

$$\texttt{sat}_\texttt{E}\texttt{(X,Y,s(J))} \leftarrow \texttt{sat}_\texttt{G}\texttt{(Y,s(J))},\ \texttt{has\_sat}_\texttt{F}\texttt{(X,J)}$$

Conjunction can also be used to handle *simultaneous* events. Consider e.g.

$$\texttt{E} = (\ \texttt{upd(A(X))},\ (\ \texttt{ins(B(Y))}\ \&\ \texttt{del(C(Z))}\ )\ ),$$

This composite event occurs when the first basic event is immediately followed by the simultaneous occurrence of the last two basic events. Its translation follows:

$$\begin{array}{ll}
\texttt{sat}_1\texttt{(X,J)} \leftarrow & \texttt{upd_A(X,J)} \\
\texttt{sat}_2\texttt{(X,Y,J)} \leftarrow & \texttt{ins_B(Y,s(J))},\ \texttt{sat}_1\texttt{(X,J)} \\
\texttt{sat}_3\texttt{(X,Z,J)} \leftarrow & \texttt{del_C(Z,s(J))},\ \texttt{sat}_1\texttt{(X,J)} \\
\texttt{sat}_\texttt{E}\texttt{(X,Y,Z,J)} \leftarrow & \texttt{sat}_2\texttt{(X,Y,J)},\ \texttt{sat}_3\texttt{(X,Z,J)}
\end{array}$$

Eventhough simultaneous events have not been discussed in previous approaches, there are many cases where this functionality is desired. As discussed in the beginning of section 3, this is necessary when *transaction boundaries granularity* must be modeled. Simultaneous events may also occur in a distributed or multiprocessor environment.

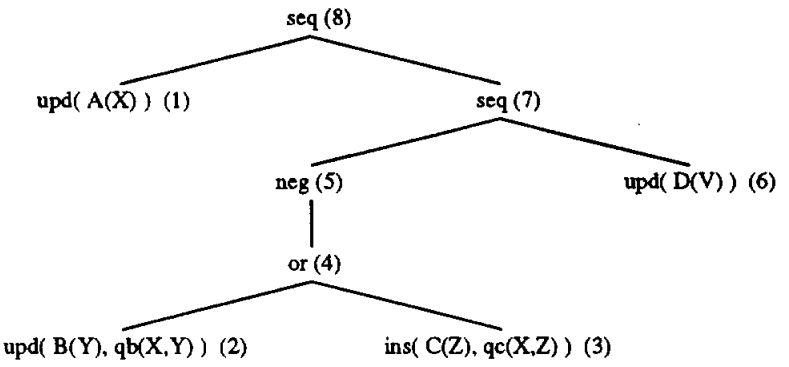

Figure 2: An EPL expression with a negated subevent

3.6 Negation

Handling negation of arbitrary composite events has been problematic in most of the previous approaches, which therefore support only limited forms of negation.

Using $Datalog_{1S}$, the semantics of general negation can be easily defined. For instance, for the negated qualified basic event
`E = !ins(ACC(X), X.Type = "Savings")`, we have (using domain variables):

$$\mathtt{sat_F}(\mathtt{Accno}, \mathtt{Owner}, \mathtt{Type}, \mathtt{Bal}, \mathtt{Time}, \mathtt{J}) \leftarrow \mathtt{ins\_ACC}(\mathtt{Accno}, \mathtt{Owner}, \mathtt{Type}, \mathtt{Bal}, \mathtt{Time}, \mathtt{J}),\ \mathtt{Type} = \text{“Savings”}$$
$$\mathtt{sat_E}(\mathtt{J}) \leftarrow \mathtt{any}(\mathtt{J}),\ \neg\mathtt{sat_F}(\_, \_, \_, \_, \_, \mathtt{J}).$$

The second rule expresses the fact that E occurs at every stage where *some* basic event occurs, but F does not occur. Referring to the `hist_monit` table, E occurs at every stage, except stage 3, where an insertion of a savings account is recorded.

As these rules show, the variables defined inside a negated event are not exported outside it. This restriction ensures the safety and domain-independence of EPL expressions.

The general case is similar. The second rule above can still be used for the negated event `E = !F`, where F is an arbitrary event. Note that at every stage that $\mathtt{sat_E}$ is satisfied, we have the occurrence of a different instance of E, and thus, every such instance has *single stage duration.*

The following example illustrates how negated composite events are handled. This example also demonstrates a disjunctive event.

Example 8 *The EPL expression for Figure 2.*
$(\ \mathtt{upd(A(X))},\ \ !\ \{\mathtt{upd(B(Y)}, \mathtt{q_b(X,Y))},\ \mathtt{ins(C(Z)}, \mathtt{q_c(X,Z))}\},\ \ \mathtt{upd(D(V))}\ )$

The satisfaction of the negated event E_5 (at a stage where neither E_2, nor E_3 occur) must intervene the occurrences of basic events E_1 and E_6.

Since a negated event instance has *single stage duration*, negated events are treated similarly to basic events, as far as *ordering* is concerned. Namely, one rule of the form shown in Table 2 is created for each of the possible predecessor of a negated event. In this example, the only possible predecessor of the negated event E_5 is E_1.

The $Datalog_{1S}$ rules for Example 8 follow:

$$
\begin{array}{ll}
\mathtt{sat}_1(\mathtt{X},\mathtt{J}) \leftarrow & \mathtt{upd\_A}(\mathtt{X},\mathtt{J}). \\
\mathtt{sat}_2(\mathtt{X},\mathtt{Y},\mathtt{s}(\mathtt{J})) \leftarrow & \mathtt{upd\_B}(\mathtt{Y},\mathtt{s}(\mathtt{J})),\ \mathtt{sat}_1(\mathtt{X},\mathtt{J}),\ \mathtt{q_b}(\mathtt{X},\mathtt{Y}). \\
\mathtt{sat}_3(\mathtt{X},\mathtt{Z},\mathtt{s}(\mathtt{J})) \leftarrow & \mathtt{ins\_C}(\mathtt{Z},\mathtt{s}(\mathtt{J})),\ \mathtt{sat}_1(\mathtt{X},\mathtt{J}),\ \mathtt{q_c}(\mathtt{X},\mathtt{Z}). \\
\mathtt{sat}_4(\mathtt{X},\mathtt{J}) \leftarrow & \mathtt{sat}_2(\mathtt{X},\mathtt{Y},\mathtt{J}). \\
\mathtt{sat}_4(\mathtt{X},\mathtt{J}) \leftarrow & \mathtt{sat}_3(\mathtt{X},\mathtt{Z},\mathtt{J}). \\
\mathtt{sat}_5(\mathtt{X},\mathtt{s}(\mathtt{J})) \leftarrow & \mathtt{any}(\mathtt{s}(\mathtt{J})),\ \mathtt{sat}_1(\mathtt{X},\mathtt{J}),\ \neg\mathtt{sat}_4(\_,\mathtt{s}(\mathtt{J})). \\
\mathtt{sat}_6(\mathtt{X},\mathtt{V},\mathtt{J}) \leftarrow & \mathtt{upd\_D}(\mathtt{V},\mathtt{s}(\mathtt{J})),\ \mathtt{sat}_5(\mathtt{X},\mathtt{J}). \\
\mathtt{sat}_7(\mathtt{X},\mathtt{V},\mathtt{J}) \leftarrow & \mathtt{sat}_6(\mathtt{X},\mathtt{V},\mathtt{J}). \\
\mathtt{sat}_E(\mathtt{X},\mathtt{V},\mathtt{J}) \leftarrow & \mathtt{sat}_7(\mathtt{X},\mathtt{V},\mathtt{J}).
\end{array}
$$

The rule for $\mathtt{E}_5$ expresses the fact that $\mathtt{E}_5$ occurs at the stage immediately following $\mathtt{E}_1$'s occurrence, if neither $\mathtt{E}_2$, nor $\mathtt{E}_3$ occur at this stage.

$\mathtt{E}_1$ is considered to be a possible predecessor of $\mathtt{E}_2$ and $\mathtt{E}_3$ as well. Generally, in a sequence expression of the form (F, !G), the subexpression G is evaluated with respect to the basic event history starting right after the satisfaction of F. Using Table 1, we get for instance:
$\mathtt{PPS}(2) = \mathtt{PPS}(4) = \mathtt{PPS}(5) = \mathtt{PPS}(7) = \{1\}$

4 Application to Other Systems

One of the most appealing characteristics of the proposed method is its generality, whereby it can be used for the formal definition of the concepts appearing in previous systems.

For the case of ODE, this is straightforward, since most of the language constructs in EPL and ODE are the same. The most important difference is that, in ODE, the *relaxed sequence* is a basic construct and the *immediate sequence* is a derivative one. Thus, the semantics of ODE can be defined by a syntax-directed translation from ODE expressions into $Datalog_{1S}$ rules, similar to that used for EPL. [3]

Such translation procedures can also be defined for Snoop and SAMOS's language, that have a somewhat different flavour from EPL and ODE. One of their differences is that the meaning of the sequence of two events $\mathtt{E}_1$ and $\mathtt{E}_2$ is equivalent to the meaning of $\mathtt{prior}(\mathtt{E}_1, \mathtt{E}_2)$ in EPL and ODE.

Because of space limitations we cannot be exhaustive and in our discussion, we focus instead on concepts which are fundamental and distinguishing. Specifically, for Snoop, we focus on its novel concept of parameter contexts and we exemplify how this can be incorporated in our method. For SAMOS, we give a very detailed example of how we can formally define the execution semantics of Petri Nets, which have been used for its implementation. For an extended treatment of Snoop and its parameter contexts, the reader is referred to [15].

4.1 Parameter Contexts

The *parameters contexts* introduced in Snoop can be used to detect and compute the parameters of composite events in different ways, and thus, they can be very useful in precisely matching the semantics of a wide range of applications

[3] The definition of disjunction in ODE seems to be problematic. In [12], only conjunction is defined as a primitive construct, while disjunction is defined through negation, via De Morgan's law. This is a problem in the presence of variables and also when disjunctions with multiple-stage duration are concerned (a negative event has always single stage duration).

[4]. The general semantics of EPL and ODE correspond to the *unrestricted context* of Snoop.

For two of Snoop's parameter contexts, we show how to extend our method, in order to adopt them. We use a *relaxed sequence* example, since the various parameter contexts arise essentially by different interpretations of such sequences.

Example 9 *The EPL expression* `E(X,Y,Z) = [A(X),B(Y),C(Z)]`, *and the event history:* $\ldots, A(1), \ldots A(2), \ldots, B(1), \ldots C(1), \ldots B(2), \ldots C(2)$, *where* `A,B,C` *denote basic events with parameters* `X,Y,Z` *respectively.*

In the *unrestricted context*, all the instances of `E` are detected. These are:
$[A(1), B(1), C(1)]$, $[A(2), B(1), C(1)]$, $[A(1), B(1), C(2)]$,
$[A(2), B(1), C(2)]$, $[A(1), B(2), C(2)]$, $[A(2), B(2), C(2)]$.

- **Recent Context.** In this context, at each stage of the history, only the most recent occurrences of the events (primitive or composite) are considered. The following instances of `E` are detected in the recent context: $[A(2), B(1), C(1)]$ and $[A(2), B(2), C(2)]$.

  We can enforce this parameter context by defining a predicate $\texttt{last\_sat}_{\texttt{F}}(\texttt{X}, \texttt{J})$, for each Snoop subexpression `F(X)`. In the $\texttt{last\_sat}_{\texttt{F}}$, for a particular stage `J`, `X` denotes the set of parameter bindings of the *last* occurrence of `F`, *before or at* this stage. The following stratified $Datalog_{1S}$ rules define the semantics of our example EPL expression, in the recent context:

$$
\begin{array}{ll}
\texttt{sat}_1(\texttt{X}, \texttt{J}) \leftarrow & \texttt{A}(\texttt{X}, \texttt{J}) \\
\texttt{last\_sat}_1(\texttt{X}, \texttt{J}) \leftarrow & \texttt{sat}_1(\texttt{X}, \texttt{J}) \\
\texttt{last\_sat}_1(\texttt{X}, \texttt{s}(\texttt{J})) \leftarrow & \texttt{any}(\texttt{s}(\texttt{J})),\ \texttt{last\_sat}_1(\texttt{X}, \texttt{J}),\ \neg\texttt{sat}_1(\_, \texttt{s}(\texttt{J})) \\
\texttt{sat}_2(\texttt{X}, \texttt{Y}, \texttt{s}(\texttt{J})) \leftarrow & \texttt{B}(\texttt{Y}, \texttt{s}(\texttt{J})),\ \texttt{last\_sat}_1(\texttt{X}, \texttt{J}) \\
\texttt{last\_sat}_2(\texttt{X}, \texttt{Y}, \texttt{J}) \leftarrow & \texttt{sat}_2(\texttt{X}, \texttt{Y}, \texttt{J}) \\
\texttt{last\_sat}_2(\texttt{X}, \texttt{Y}, \texttt{s}(\texttt{J})) \leftarrow & \texttt{any}(\texttt{s}(\texttt{J})),\ \texttt{last\_sat}_2(\texttt{X}, \texttt{Y}, \texttt{J}),\ \neg\texttt{sat}_2(\_, \_, \texttt{s}(\texttt{J})) \\
\texttt{sat}_{\texttt{E}}(\texttt{X}, \texttt{Y}, \texttt{Z}, \texttt{s}(\texttt{J})) \leftarrow & \texttt{C}(\texttt{Z}, \texttt{s}(\texttt{J})),\ \texttt{last\_sat}_2(\texttt{X}, \texttt{Y}, \texttt{J})
\end{array}
$$

- **Chronicle Context.** In the chronicle context, when a composite event `E` is satisfied, its parameter bindings are obtained from the oldest *unused* occurrences of its component events, that satisfy the precedence requirements of `E` (*unused* implying that the same basic event occurrence can participate in *at most one* instance of a particular composite event).

  Thus, the two instances of `E` that would be detected in the chronicle context are: $(A(1); B(1); C(1))$ and $(A(2); B(2); C(2))$.

  A way to express the chronicle context semantics using $Datalog_{1S}$ is illustrated in the next section, since as it is explained there, the execution semantics of Petri Nets correspond to this parameter context.

4.2 SAMOS and Petri Nets

The most distinguishing feature of SAMOS is its event detection mechanism, which is based on *coloured Petri Nets* [6]. A *coloured Petri Net* is an extended version of a classical Petri Net, that allows the flow of parameter bindings

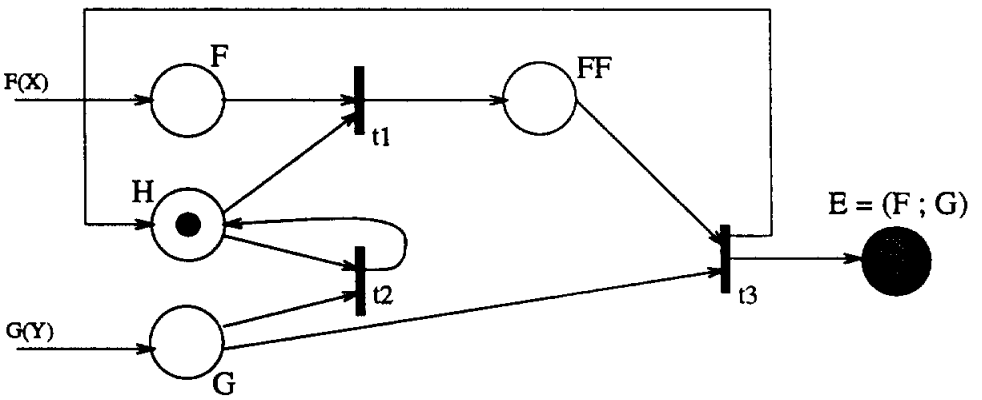

Figure 3: The Petri Net for the expression E = (F ; G).

through it. In this way, the parameter passing within a composite event instance can be modeled. For a detailed description of how Petri Nets can be used for the detection of composite events, the reader is referred to [11]. Here we focus on the example of Figure 3 borrowed from [10], which illustrates the basic concepts. The figure shows the initial state of the Petri Net, where a token is contained in the *auxiliary place* H.

This Petri Net accepts as input the occurrences of events F and G. Every such occurrence corresponds to a new token inserted into the Petri Net; this token contains the parameter bindings of the event occurrence. An output token is created when an occurrence F(x) is matched with an appropriate occurrence G(y). The output token containing the parameter bindings (x, y) is deposited into the output place E and a new occurrence of E is signaled. Right after that, this output token is removed from the Petri Net.

At a particular point in time, the tokens of the unmatched occurrences of F are contained in places F and FF. In particular, FF contains the token of the oldest among these occurrences, where F contains the tokens for the rest of them. These different tokens are distinguished by their *colours*. Specifically, a token corresponding to an occurrence of F is *coloured* with an integer denoting the *order* of this occurrence among all the occurrences of F in the event history (this can be obtained by a method similar to that defining the *stage* of a basic event in the event history).

An occurrence of G is ignored, if there is not any token (occurrence) in FF to be matched with; observe how this is achieved by using the *auxiliary place* H. However, an occurrence G(y) that finds a token F(x) in FF is matched with it (this is achieved by firing the gate t3) and the occurrence of E(x, y) is then signaled. After that, H becomes active again (it contains a token) and if there are any tokens in F, the token of the *oldest* unmatched occurrence of F is passed to FF (through the firing of gate t1), so that it can be matched with the next occurrence of G. Note that this behaviour corresponds to the *chronicle* parameter context discussed in the previous section. The execution semantics of our Petri Net is best illustrated by an example, such as the one of Table 3.

Since, places F and FF are *coloured*, the colours (order numbers) of their tokens are shown as well. Place H is also coloured and the colour of its token is N+1, if N occurrences of E have been signaled so far (and thus, the first N occurrences of F have been used).

In coloured ordinary Petri Nets there is not inherent measure of time, or

Event	H	F	FF	G	E
	1	-	-	-	-
$G(y_a)$	1	-	-	y_a	-
t_2 fires	1	-	-	-	-
$F(x_a)$	1	$(x_a,1)$	-	-	-
t_1 fires	-	-	$(x_a,1)$	-	-
$F(x_b)$	-	$(x_b,2)$	$(x_a,1)$	-	-
$G(y_b)$	-	$(x_b,2)$	$(x_a,1)$	y_b	-
t_3 fires	2	$(x_b,2)$	-	-	(x_a,y_b)
t_1 fires	-	-	$(x_b,2)$	-	-
$G(y_c)$	-	-	$(x_b,2)$	y_c	-
t_3 fires	3	-	-	-	(x_b,y_c)

Table 3: An execution sequence of the Petri Net in Figure 3.

model of time flow. Thus, ordinary Petri Nets make it possible to describe *what* happens, but not *when* it happens [6]. Aiming at resolving this issue, an extension of ordinary Petri Nets, called *synchronized* Petri Nets, has recently been proposed. They enable the modeling of systems whose evolutions are time dependent and where the succession of system *states* must be clearly and deterministically described. The *state* of a Petri Net is defined by the assignment of tokens to places.

In a synchronized Petri Net, gate firings are synchronized to external events and they obey a particular partial order defined by the structure of the Petri Net. An external event triggers a sequence of gate firings that occur in one or more *steps*, until the Petri Net reaches a *stable state*. This is exemplified in Table 3.

We now demonstrate how $Datalog_{1S}$ can be used to capture the execution semantics of a synchronized Petri Net. We assume that the occurrences of basic events are *non-simultaneous*, as required by Petri Nets (this is in fact a limitation of theirs). The following $Datalog_{1S}$ program defines the execution semantics of the Petri Net in Figure 3.

$$
\begin{array}{ll}
\mathtt{in\_F}(\mathtt{X},\mathtt{N},\mathtt{J}) \leftarrow & \mathtt{F}(\mathtt{X},\mathtt{N},\mathtt{J}) \\
\mathtt{in\_F}(\mathtt{X},\mathtt{N},\mathtt{s}(\mathtt{J})) \leftarrow & \mathtt{any}(\mathtt{s}(\mathtt{J})),\ \mathtt{in\_F}(\mathtt{X},\mathtt{N},\mathtt{J}),\ \neg \mathtt{t}_1(\mathtt{X},\mathtt{N},\mathtt{s}(\mathtt{J})) \\
\mathtt{in\_H}(1,0) & \\
\mathtt{in\_H}(\mathtt{N},\mathtt{s}(\mathtt{J})) \leftarrow & \mathtt{any}(\mathtt{s}(\mathtt{J})),\ \mathtt{in\_H}(\mathtt{N},\mathtt{J}),\ \neg \mathtt{t}_1(\_,\mathtt{N},\mathtt{s}(\mathtt{J})),\ \neg \mathtt{t}_2(\_,\mathtt{N},\mathtt{s}(\mathtt{J})) \\
\mathtt{in\_H}(\mathtt{N},\mathtt{J}) \leftarrow & \mathtt{t}_2(\_,\mathtt{N},\mathtt{J}) \\
\mathtt{in\_H}(\mathtt{N}+1,\mathtt{J}) \leftarrow & \mathtt{t}_3(\_,\_,\mathtt{N},\mathtt{J}) \\
\mathtt{in\_G}(\mathtt{Y},\mathtt{J}) \leftarrow & \mathtt{G}(\mathtt{Y},\mathtt{J}) \\
\mathtt{in\_G}(\mathtt{Y},\mathtt{s}(\mathtt{J})) \leftarrow & \mathtt{any}(\mathtt{s}(\mathtt{J})),\ \mathtt{in\_G}(\mathtt{Y},\mathtt{J}),\ \neg \mathtt{t}_2(\mathtt{Y},\_,\mathtt{s}(\mathtt{J})),\ \neg \mathtt{t}_3(\_,\mathtt{Y},\_,\mathtt{s}(\mathtt{J})) \\
\mathtt{in\_FF}(\mathtt{X},\mathtt{N},\mathtt{J}) \leftarrow & \mathtt{t}_1(\mathtt{X},\mathtt{N},\mathtt{J}) \\
\mathtt{in\_FF}(\mathtt{X},\mathtt{N},\mathtt{s}(\mathtt{J})) \leftarrow & \mathtt{any}(\mathtt{s}(\mathtt{J})),\ \mathtt{in\_F}(\mathtt{X},\mathtt{N},\mathtt{J}),\ \neg \mathtt{t}_3(\mathtt{X},\_,\mathtt{N},\mathtt{s}(\mathtt{J})) \\
\mathtt{sat}_{\mathtt{E}}(\mathtt{X},\mathtt{Y},\mathtt{J}) \leftarrow & \mathtt{t}_3(\mathtt{X},\mathtt{Y},\_,\mathtt{J}) \\
\mathtt{t}_1(\mathtt{X},\mathtt{N},\mathtt{s}(\mathtt{J})) \leftarrow & \mathtt{in\_F}(\mathtt{X},\mathtt{N},\mathtt{J}),\ \mathtt{in\_H}(\mathtt{N},\mathtt{J}) \\
\mathtt{t}_2(\mathtt{Y},\mathtt{N},\mathtt{s}(\mathtt{J})) \leftarrow & \mathtt{in\_G}(\mathtt{Y},\mathtt{J}),\ \mathtt{in\_H}(\mathtt{N},\mathtt{J}) \\
\mathtt{t}_3(\mathtt{X},\mathtt{Y},\mathtt{N},\mathtt{s}(\mathtt{J})) \leftarrow & \mathtt{in\_FF}(\mathtt{X},\mathtt{N},\mathtt{J}),\ \mathtt{in\_G}(\mathtt{Y},\mathtt{J})
\end{array}
$$

An *order parameter* N for the occurrences of basic event F has been introduced. A different predicate is defined for each place and each gate.

Consider for instance the rules for place F. The first rule says that the token for the *N-th* occurrence of F is deposited into the place of F, at the stage of that occurrence. The second rule says that the token of *colour N* (corresponding to the *N-th* occurrence of F) is contained in place F at stage s(J), if it is contained there at stage J and gate t1 is not enabled at stage s(J), which would cause that token to be removed.

On the other hand, the rule for a gate expresses the fact that if all its inputs become active at stage J, then the gate is enabled at stage s(J). An output token is created from information passed from the inputs of gate t3 and it is deposited into the place for E — see the rules for t3 and $\mathtt{sat}_{\mathtt{E}}$.

The *component by component* translation described above can produce a lengthy set of rules. Moreover, note that we have lost the direct correspondence between basic event occurrences and stages, which we have assumed in the previous sections.

We can solve these problems by providing a much simpler set of $Datalog_{1S}$ rules, that still models precisely the execution semantics of our example Petri Net and also maintains the one-to-one correspondence between external basic events and stages. This is achieved by folding the sequence of stages (states) in Table 3 into the one in Table 4.

In the new table, each external basic event creates only one new stage. This stage is defined by (a) the tokens residing at the regular places F and FF, at the end of the *sequence of firings* originated by the basic event, and (b) the token (if any) deposited into the output place E, during this sequence of firings.

Event	F	FF	E
	-	-	-
$G(y_a)$	-	-	-
$F(x_a)$	-	$(x_a,1)$	-
$F(x_b)$	$(x_b,2)$	$(x_a,1)$	-
$G(y_b)$	-	$(x_b,2)$	$(x_a y_b,1)$
$G(y_c)$	-	-	$(x_b y_c,2)$

Table 4: The simplified stage sequence for the execution sequence of Table 3.

The following $Datalog_{1S}$ rules defines the sequence of stages in Table 4:

$$\mathtt{in\_F(X,N,s(J))} \leftarrow \quad \mathtt{F(X,N,s(J))},\ \mathtt{in\_FF(\_,\_,J)}$$
$$\mathtt{in\_F(X,N,s(J))} \leftarrow \quad \mathtt{any(s(J))},\ \mathtt{in\_F(X,N,J)},\ \neg\mathtt{in\_FF(X,N,s(J))}$$
$$\mathtt{in\_FF(X,N,s(J))} \leftarrow \quad \mathtt{F(X,N,s(J))},\ \neg\mathtt{in\_FF(\_,\_,J)}$$
$$\mathtt{in\_FF(X,N+1,s(J))} \leftarrow \quad \mathtt{in\_F(X,N+1,J)},\ \mathtt{sat_E(\_,\_,N,s(J))}$$
$$\mathtt{sat_E(X,Y,N,s(J))} \leftarrow \quad \mathtt{G(Y,s(J))},\ \mathtt{in\_FF(X,N,J)}$$

Note that no predicate is needed for auxiliary place H. Also, no predicate is needed for G, since an occurrence of G is either immediately matched with an occurrence of F, or it is "dumped". As a result, we obtain a much simpler set of rules that still defines precisely the execution semantics of our example Petri Net.

Eventhough these rules are not stratified, they are locally stratified and *XY-stratified* [19]; therefore they can be evaluated in an incremental and efficient manner. To see that, observe that the computation of the contents of `in_F` and `in_FF` at stage `J` can be followed by the computation of the contents of `in_FF` at stage `s(J)`, which can then be followed by the computation of the contents of `in_F` at stage `s(J)`. This order of evaluation (prescribed by the XY-stratification of the program) leads to an efficient fixpoint, which is triggered by each basic event occurrence and terminates in a fixed number of steps.

5 Related Work

There has been no generally accepted approach to the definition of semantics and to the implementation of event detection mechanisms for composite event specification languages; each system employs a different formalism.

We consider first the familiar model of Finite State Machines (FSMs), which is the implementation framework of ODE. Eventhough FSMs provide an easy-to-understand model that is suggestive of efficient implementation, they suffer from two major limitations, as follows:

1. FSMs do not support variables. It is suggested in [12] that parameterized events are handled by creating several instances of an FSM, one for each set of partial variable bindings of the composite event that the FSM implements (detects). The resulting model surrenders the initial simplicity and intuitive appeal of FSMs, without providing a fail-proof formalism. In particular, the semantics of negation when attributes are involved remains a problem.
2. Since FSMs are inherently sequential, simultaneous events cannot be handled, unless transitions based on combinations of events are allowed. But even if simultaneous events are disallowed, constructs such as conjunction can only be modeled by an exponentially increasing size of states in an FSM. This is because the FSM for $E = E_1 \wedge E_2$ is built by constructing the cross-product of the FSMs for E_1 and E_2 [14]. Instead, in EPL, the fact that the two conjuncts are evaluated *in parallel* and *independently* from each other leads to the generation of a number of rules that equals the sum of the numbers of rules generated for the two conjuncts, plus an extra rule for the conjunction condition.

Petri Nets solve the problem of exponential blow-up in FSMs, by allowing concurrent processing. Also, coloured Petri Nets cater for the handling of parameterized composite events. However, there are still some limitations. Simultaneous events cannot be handled. Also, it is not clear how the semantics of general negation (as defined in EPL) can be captured by Petri nets. The composite event detection mechanism of Snoop, which is based on Event Graphs, [3] suffers from these limitations as well.

Another concept, whose formal definition is missing in previous approaches is that of *event histories* and the *succession of states* in the system. This becomes necessary when introducing simultaneous events. Finally, all previous approaches are limited to operational specifications, where our logic-based approach has a declarative nature.

Our method has similarities to Chomicki's work on the efficient detection of violations of dynamic integrity constraints [5], which is not however directly

related to general active database systems. Integrity constraints are expressed in Temporal Logic. Another difference of [5] from our work is that *condition-action* rules are used, which are re-evaluated at each stage, in an incremental, *history-less* way. This is contrasted to our *event-driven* rules.

6 Conclusions and Future Work

In this paper, we have proposed a general method for the definition of the semantics of composite event languages, such as those used in active temporal databases. Whereas different formalisms were used in the past for different languages, we have shown here that the same formalism can be used to define the concepts of every language. As a result, the task of comparing and understanding the differences and similarities of the languages is simplified.

Besides generality, another advantage of the proposed approach is that event histories, simultaneous events, variables and their scopes, parameter contexts, negation, conjunction and disjunction can all be part of the same model. $Datalog_{1S}$ is a most natural tool for defining the semantics of event languages, due to its ability to model both the temporal and logical aspects of queries.

From an implementation standpoint, the produced sets of rules can be efficiently evaluated in an incremental and history-less manner.

This paper leaves several issues to further research. For instance, the issue of comparing the expressive power of different composite event languages is one that deserves much more attention than it has received so far. In this respect, $Datalog_{1S}$ provides a sound formal basis, due to the fact that its formal semantics is well-understood and its expressive power w.r.t. to other languages (temporal or otherwise) has been previously characterized [2].

A related issue is to compare the effectiveness of different temporal reasoning formalisms in expressing the semantics of active temporal languages. For instance, composite event expressions could also be translated into Temporal Logic [8], or a temporal logic programming language, such as Templog [2], or SimTL [18]. In many cases, this yields a very natural formal definition of the semantics such expressions. For instance, the following *Future Temporal Logic* formula defines the meaning of the EPL expression `E = (A, *:(*:B, C), D)`, which has a form similar to that of example 7:

$$E = A \ \textbf{next} \ (\ (B \ \textbf{until} \ C) \ \textbf{until} \ D\ )$$

On the other hand, $Datalog_{1S}$ is normally more conducive to effective operational semantics than Temporal Logic formulas. Thus, one might prefer to use $Datalog_{1S}$, or alternatively, to derive efficient operational semantics by translating Temporal Logic formulas into $Datalog_{1S}$, a mapping discussed in [2]. Finally, the optimization of operational semantics presents many opportunities for significant improvements and interesting research.

References

[1] N. Arni, K. Ong, S. Tsur, and C. Zaniolo. LDL++: A second generation deductive database system. submitted for publication.

[2] M. Baudinet, J. Chomicki, and P. Wolper. Temporal Deductive Databases. In A. Tansel et al., editor, *Temporal Databases: Theory, Design and Implementation*, chapter 13, pages 294–320. Benjamin/Cummings, 1993.

[3] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S. K. Kim. Anatomy of a composite event detector. Technical Report CIS TR-93-039, University of Florida, December 1993.

[4] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S. K. Kim. Composite events for active databases: Semantics, contexts and detection. In *Proceedings of the 20th VLDB Conference*, pages 606–617, September 1994.

[5] J. Chomicki. History-less checking of dynamic integrity constraints. In *Proceedings of the International Conference on Data Engineering*, pages 557–564, 1992.

[6] R. David. *Petri Nets and Grafcet: Tools for modeling discrete event systems.* Prentice Hall, New York, 1992.

[7] U. Dayal, E.N. Hanson, and J. Widom. Active Database Systems. In W. Kim, editor, *Modern Database Systems.* Addison Wesley, 1995.

[8] E. Emerson. Temporal and Modal Logic. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 16, pages 995–1072. Elsevier/MIT Press, 1990.

[9] U. Dayal et al. The HiPAC Project: Combining active databases and timing constraints. *ACM-SIGMOD Record*, 17(1):51–70, March 1988.

[10] S. Gatziu and K. R. Dittrich. Events in an object-oriented database system. In *Proceedings of the First Intl. Conference on Rules in Database Systems*, pages 23–39, September 1993.

[11] S. Gatziu and K. R. Dittrich. Detecting composite events in active databases using petri nets. In *Proceedings of the 4th Intl. Workshop on Research Issues in Data Engineering: Active Database Systems*, pages 2–9, 1994.

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

[13] N. H. Gehani, H. V. Jagadish, and O. Shmueli. Event specification in an active object-oriented database. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 81–90, 1992.

[14] D. Harel, A. Pnueli, J. P. Schmidt, and R. Sherman. On the formal semantics of statecharts. In *Proceedings of the 2nd IEEE Symposium on Logic in Computer Science*, pages 54–64, 1987.

[15] I. Motakis and C. Zaniolo. Composite Temporal Events in Active Database Rules: A Logic-Oriented Approach. submitted for publication.

[16] NASA, Lyndon Johnson Space Center, Software Technology Branch. *CLIPS 6.0 Reference Manual*, June 1993.

[17] N. Pissinou, R. Snodgrass, R. Elmasri, I. Mumick, M. Ozsu, B. Pernici, A. Segev, B. Theodoulidis, and U. Dayal. Towards an infrastructure for temporal databases. *ACM-SIGMOD Record*, 23(1), March 1994.

[18] A. Tuzhilin. Applications of Temporal Databases to Knowledge-based Simulations. In A. Tansel et al., editor, *Temporal Databases: Theory, Design and Implementation*, chapter 23. Benjamin/Cummings, 1993.

[19] C. Zaniolo. A unified semantics for active and deductive databases. In *Proceedings of the 1st International Workshop on Rules in Database Systems*, pages 271–287, 1993.