JOSI, , 1–37 (1997)

Formal Semantics for Composite Temporal Events in Active Database Rules*

IAKOVOS MOTAKIS** imotak@ctp.com
Cambridge Technology Partners, 2361 Rosecrans Ave., El Segundo, CA 90245

CARLO ZANIOLO zaniolo@cs.ucla.edu
University of California, Los Angeles, CA 90024

Editor: Peter Ng

Abstract. A major thrust of current research in active databases focuses on allowing complex patterns of temporal events to serve as preconditions for rule triggering. Currently, there is no common formalism for specifying the semantics of composite event languages. Different systems have used an assortment of different techniques, including Finite State Automata, Petri Nets and Event Graphs. In this paper, we propose a unifying approach, based on a syntax-directed translation of composite event expressions into $Datalog_{1S}$ rules, whose formal semantics defines the meaning of the original expressions. We demonstrate our method by providing a formal specification of the Event Pattern Language (EPL) developed at UCLA. This method overcomes problems and limitations affecting previous approaches and is applicable to other languages such as ODE, SNOOP and SAMOS—thus, allowing a more direct comparison across different systems.

Keywords: active databases, temporal databases, deductive databases, composite events

1. Introduction

A new generation of database systems support active rules for the detection of events occurring in the database and the triggering of induced actions. In many applications, however, the simple-event detection capabilities found in commercial systemsare not sufficient; complex sequences of events must instead be detected as the natural preconditions for taking actions and firing rules [9, 34]. Sophisticated research prototypes have in fact been recently implemented to provide this capability— an incomplete list includes [9, 16, 13, 5].

A difficult problem facing the designers of active databases is that of defining the semantics of active rule languages. Indeed, assigning a formal semantics to active database behavior presents a challenge even when only simple events are involved [28, 27]. For composite event languages, this problem becomes even more complex, inasmuch as the issues of temporal databases and active rule languages are now combined and intertwined [29].

In previous systems, the operational semantics of composite event detection has been defined using widely different frameworks, such as Finite State Automata [15], Petri Nets [14], or Event Graphs [4]. Even though these systems have captured many aspects of the functionality required by active database applications, they

* Work supported by NSF Grant IRI-9632272
** This work was performed at the University of California, Los Angeles

are restricted by the limitations of their underlying formalisms. Furthermore, the idiosyncrasies of these formalisms make the languages difficult to compare. Consequently, it would be desirable to have a more general formalism that facilitates correctness testing, as well as optimization and extensibility studies.

By contrast, powerful semantic foundations are available for the rules of deductive databases [1, 21]; so far, however, these have had little impact on the design of active rule languages. In this paper, we show that this omission was quite unfortunate, since $Datalog_{1S}$, a temporal extension of *Datalog*, provides a natural and powerful basis for defining the semantics of composite-event-triggered database rules. The general method we propose is based on a syntax-directed translation of such active rules into $Datalog_{1S}$ rules, whose formal semantics defines the meaning of the original rules.

We demonstrate the method by presenting a complete definition of Event Pattern Language (EPL). EPL is the rule language of an active database system designed and implemented at UCLA, which provides the capability of detecting and acting upon complex patterns of events that evolve over time. The paper also explains how this method can be applied to the composite event languages of ODE [16], SNOOP [5] and SAMOS [13], in order to define their formal semantics and compare their features.

The proposed method matches and surpasses the capabilities of the approaches used in the past, in terms of defining the semantics of different language characteristics, including *event attributes, negation, simultaneous events, and parameter contexts*. Thus, we obtain a general rigorous method for specifying the semantics of composite-event languages, and a useful tool for comparing their constructs and their expressive power.

2. Motivations

Among the various formalisms used in the past for implementing composite event detectors, that of Finite State Automata (FSA) used in ODE is probably the simplest and most familiar one.

Figure 1 shows a sample *event history* and a FSA that can recognize the *first* occurrence of a composite event. Here, `f` and `g` denote two basic events and `(f,g)` denotes an immediate sequence, i.e., event `g` must immediately follow event `f`.

In ODE, for each composite event expression `E`, a general *non-deterministic* automaton is created, which can succeed multiple times, once for each occurrence of `E`. Unfortunately, when it comes to detecting composite events, FSA suffer from the following limitations:

1. *Parameterized Events.* FSA do not support variables, and thus, they cannot detect parameterized events, which are essential in most real-life applications. It is suggested in [15] that parameterized events can be handled by using relational automata. A relational automaton is a FSA, with the addition that each one of its states is augmented with data structures that maintain partial variable bindings for the composite event that the FSA implements (detects). However,

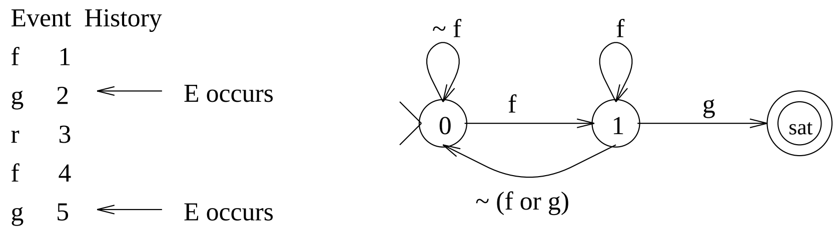

Figure 1. A FSA for the composite event E = (f, g). 0 and *sat* denote the *start* and *success* states respectively.

the resulting model surrenders the initial simplicity and intuitive appeal of FSA, without providing a fail-proof formalism for the detection of composite events. For instance, the semantics of negation when attributes are involved remains a problem.

2. *Simultaneous Events.* FSA are inherently sequential, and thus, simultaneous events cannot be supported, unless transitions based on combinations of events are allowed. This could result in a multitude of states and transitions, that would make the FSA very complicated.

3. *Exponential Blow-Up.* In ODE, constructs that require the concurrent evaluation of their components can cause an exponential blow-up in the size (number of states) of the resulting FSA. Consider for instance the *conjunction* event $\mathtt{E} = \mathtt{E}_1 \wedge \mathtt{E}_2$. Its FSA is built by constructing the cross-product of the FSAs for $\mathtt{E}_1$ and $\mathtt{E}_2$.

A natural step beyond FSA is to use Statecharts [19, 18], which offer an appealing graphical formalism. Statecharts provide the ability to model constructs that require the concurrent evaluation of (sub)event expressions, without an exponential increase in the number of required states. However, Statecharts suffer from limitations in regard to their expressive power, the most notable of which being their lack of support for parameterized events [19]. Due to these limitations and the fact that they have not been used by any previous system, we will not discuss Statecharts any further.

Another formalism that has been used for defining composite events is that of Petri Nets [14]. Since Petri Nets allow concurrent processing, they do not suffer from the exponential blow-up problem mentioned above. Furthermore, *colored Petri Nets* [8] 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 negated events (as defined in the next section) can be captured by Petri nets.

The Event Graphs framework that is used in SNOOP [4] suffers from limitations similar to those of Petri Nets.

A simpler approach to overcome the mentioned problems is that of using $Datalog_{1S}$ [3]. This is a temporal deductive language that extends *Datalog*, by allowing every predicate to have (at most) one temporal argument in addition to the usual data parameters. The terms of this argument range over the assumed temporal domain, and can be constructed using the unary successor function s. For our purposes, the temporal argument can be used to model the succession of states in the event history, and therefore, we call it the *stage* argument.

For instance, all occurrences of pattern `E = (f,g)` can be detected by the following two $Datalog_{1S}$ rules:

EXAMPLE 1: $Datalog_{1S}$ rules detecting occurrences of `E = (f,g)`.

$$\begin{array}{ll} \mathtt{in_1(J)} \leftarrow & \mathtt{evt\_hist(f,J).} \\ \mathtt{sat_E(s(J))} \leftarrow & \mathtt{evt\_hist(g,s(J)),\ \ in_1(J).} \end{array}$$

□

The rules are applied against an *event history* relation **evt_hist**, where each event is sequentially numbered, as shown in Figure 1. Observe that predicate $\mathtt{sat_E}$ is satisfied at every stage where the pattern `(f,g)` occurs. Furthermore, it is trivial to support data parameters, by simply adding parameter columns to the *event history* relation, and corresponding variables to the $Datalog_{1S}$ predicates. For instance, one such parameter could be a *transaction-time timestamp* denoting when basic events occur; this facilitates the use of time-based conditions in composite event expressions. Thus, composite events evolving in time, as well as conditions over their parameter values can be easily supported.

In a nutshell, $Datalog_{1S}$ provides a simple and powerful formalism, where all the ingredients needed for the definition of composite event patterns are already in place. One key ingredient is in fact the extraordinary semantics foundations of *Datalog*, where along with an abstract least-model semantics, there exists an equivalent fixpoint-based semantics and various operational improvements, such as differential fixpoint and magic sets. Therefore, $Datalog_{1S}$ can be used as the basis for a general method for defining composite event languages and as a tool for understanding and comparing the constructs and expressive power of such languages.

The method here proposed has some similarities to Chomicki's work on specifying complex integrity constraints [7]. In this work, constraints are expressed in Temporal Logic and are translated into *condition-action* rules. A more recent approach is that of [31], which employs another variation of Temporal Logic, where assignment operators are used instead of first-order quantifiers. In [31], temporal aggregates are also considered. The *condition-action* rules used in these approaches are contrasted to our *event-driven* rules.

3. The Event Pattern Language (EPL)

EPL is an active rule language designed and implemented at U.C.L.A., that supports the specification of composite events. *Composite* events are constructed from *simple* events, using various constructs, such as sequence, repetition, conjunction, disjunction and negation. EPL has a rule-based syntax and has been designed as a portable front-end to active relational databases supporting only simple event detection. Two prototypes of EPL were developed, the first in $\mathcal{LDL}$++ [2] and the second in CLIPS [26].

In the rest of this Section, we first describe the basic components of the EPL rule language, and then, we present its composite event constructs. In our examples, we use the following bank accounts relation:

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

3.1. EPL Components

3.1.1. Basic Events. According to the practice in most commercial relational DBMS that support triggers, the *basic events* of EPL are database table modification commands. Thus, a *basic event type* is either:

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

where `Rname` is the name of a database relation. Basic events are assumed to occur instantaneously, i.e., they have a duration that is either zero or less than the basic time granule. Each basic event occurrence is time-stamped by the underlying database system.

Unlike all previous composite event languages, EPL can handle simultaneous events, i.e., multiple events with the same timestamps. Simultaneous events may naturally occur in a distributed or multiprocessor environment, or even in a single-processor system that uses coarse time granularity. Furthermore, there are situations where a set of basic events needs to be processed *simultaneously*, even if these events do not physically occur at the same time point. A typical such case is when the *transaction granularity* is used for basic event processing. Under this scheme, the processing of all basic events identified within a transaction is deferred until the end of the transaction, just before its commit point. All these basic events are then assigned the same timestamp, which is usually the transaction's commit time.

3.1.2. Simple Event Expressions. We use the term *simple event* to refer to a basic event that is optionally qualified by conditions that are evaluated when the basic event occurs. In general, a simple event expression has the form:

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

The condition expression is built using the standard arithmetic and comparison operators and the logical connectives *AND (or comma)*, *OR* and *NOT*. [1]

X is a tuple variable that corresponds to the modified tuple in relation *Rname*. Its components are the attributes in the schema of *Rname*, plus an extra attribute *Timestamp* that contains the date and time of this modification. 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, *new* is assumed.

As an example, the following qualified basic event denotes a large withdrawal from a "Savings" account:

```
update( ACC(X), X.Type = "Savings", X.old_Balance-X.new_Balance > 100,000 )
```

3.1.3. Actions. Two kinds of actions are supported in EPL: (a) SQL commands, and (b) Operating system calls, that facilitate the communication of EPL with the user, or other applications that run concurrently. For instance, the Unix *write* system call could be used as the action of the last simple event:

```
    -> write( "Large withdrawal from account %d at time %s \n",
              X.Accno, X.Timestamp ).
```

The action of a rule can use the parameters appearing in basic events in the rule's body.

3.1.4. EPL Modules. EPL rules are organized in *modules*, which can be compiled and enabled independently. The *events declaration section* of a module defines the basic event types monitored by the module. This is the *universe of events*, under which the module's rules are evaluated.

The *rules section* contains the rules themselves. Each rule has a name, which is unique within its module. A rule's body (head) corresponds to an event (action). Example 2 demonstrates an EPL rule within a module.

EXAMPLE 2: An EPL module with a rule that keeps track of large withdrawals from "Savings" accounts.

```
Begin Module ACC_Mod
   monitor  insert(ACC), update(ACC), delete(ACC);
   LargeWithdrawal:
      update( ACC(X),
              X.Type = "Savings", X.old_Balance-X.new_Balance > 100,000 )
      -> write( "Large withdrawal from account %d at time %s \n",
                X.Accno, X.Timestamp ).
end Module.
```

□

This module keeps track of all the modifications in the `ACC` relation. Rule `LargeWithdrawal` triggers upon a simple event, which is a large withdrawal from a "Savings" account.

3.2. EPL Composite Event Expressions

The EPL language allows the specification of *composite events*, that describe patterns of *simple events.*

3.2.1. Basic Notions. We distinguish between *event expressions* (also called *event types*) and *event instances.* An event expression `E` defines an event pattern of interest, which is evaluated with respect to the *event history* of a module. An instance of E consists of a set of simple event occurrences that satisfies this pattern. In the sequel, we will refer to *events* when the distinction is clear from the context.

The first simple event in an instance of E defines the beginning of this instance. Likewise, the last simple event in an instance of E defines the completion of this instance. We say that event expression E is *satisfied*, when an instance of E is completed. We will also use the term *occurrence of event* E to refer to the time point that an instance of E is completed.

Since simple events can have parameters, the same holds for composite events. The variables of a composite event are obtained from the attributes of its component simple events, according to the scope rules of the language. From now on, unless otherwise indicated, when we refer to variables, we mean *tuple variables.*

Finally, we will refer to the *universe of events*, with the understanding that this is with respect to a particular module, where the event expression appears.

3.2.2. Composite Event Constructs. The definition of composite event expressions follows. Many of the language constructs are demonstrated in Figure 2. The sequence of marked points on the top of the Figure corresponds to the sequence of time instants where simple events of interest have occurred.

Definition 1. The following are the basic EPL constructs. The language has the *closure* property, i.e., a composite event can be used as an operand of any construct.

1. An *immediate sequence* $E(X_1, X_2, \ldots, X_n) = (E_1(X_1), E_2(X_2), \ldots, E_n(X_n))$ is satisfied by an instance of E_1, *immediately* followed by an instance of E_2, ..., *immediately* followed by an instance of E_n.

   This means that an instance of E_2 begins at the next point in the event history after an instance of E_1 ends, and so on. An immediate sequence example is illustrated in Figure 2(a).

2. A *star sequence* $E(X) = *F(X)$ is satisfied by an immediate sequence of *one or more* consecutive instances of F.

   This is illustrated in Figure 2(b).

3. The *conjunction* $E(X_1, X_2, \ldots, X_n) = (E_1(X_1)\ \&\ E_2(X_2)\ \&\ \ldots\ \&\ E_n(X_n))$ occurs when all events $E_1, \ldots, E_n$ occur *simultaneously.*

This means that an instance of each one of $E_1, \ldots, E_n$ is completed at the same point in the event history. As shown in Figure 2(c), these instances may have started at different time points.

4. The *disjunction* $E = \{E_1(X_1),\ E_2(X_2),\ \ldots,\ E_n(X_n)\}$ occurs when at least one event among $E_1, \ldots, E_n$ occurs.

   An example is shown in Figure 2(d).

5. The *negated event* $E\ = !F(X)$ is satisfied (occurs) at every point in the event history, where F is not satisfied at.

   Consider for instance Figure 2(e). There are two occurrences of F in the event history. Consequently, $!F$ is satisfied at every point in the event history, except the two points where F is satisfied at.

A number of additional (derived) constructs can be defined in terms of the basic ones (see also [15]). These operators do not add to the expressive power of the language, but simplify the specification of complex event patterns. The definitions for some important derived constructs follow:

- *any.*
  The disjunction of all basic events in the universe. It occurs at every point in the event history.

- *Relaxed Sequence:* $E(X_1, X_2, \ldots, X_n) = [E_1(X_1),\ E_2(X_2),\ \ldots,\ E_n(X_n)] \equiv (E_1(X_1),\ *any,\ E_2(X_2),\ *any,\ \ldots,\ *any,\ E_n(X_n))$.

  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 Operator.*
  $E(X_1, X_2) = prior(E_1(X_1),\ E_2(X_2)) \equiv (E_1(X_1),\ *any)\ \&\ E_2(X_2)$.

  This event is illustrated in Figure 2(f). Its meaning is that an occurrence of E_2 must follow an occurrence of E_1, i.e., an instance of E_1 must be completed prior to the completion of an instance of E_2. Note that it is not required that the instance of E_2 starts after the end of the instance of E_1; this is the difference from the immediate sequence construct.

Examples of EPL composite event expressions follow.

EXAMPLE 3: [Immediate Sequence] 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 > 100,000),
     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 consists of an update to decrease the balance in the first account, followed immediately by another update to increase the balance in the second one. Thus, the order in which the events are written in the rule corresponds to the temporal order in which they occur. The condition expression in the second event can refer to the tuple variable of the first event, but not vice-versa, mimicking the natural process of memorizing values of past events in an evolving history.

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).
```

□

EXAMPLE 5: [Simple Negation] Consider the following rule, as part of module `ACC_Mod` in Example 2.

```
NoNewCheckingAcc:
     ! insert( ACC(X), X.Type = "Checking" )
     -> write("Any event, except for a creation of a new Checking account").
```

□

This rule is triggered on any basic event monitored by module `Cust_Mod`, except for small "buys" or "sells" of stocks.

EPL can also express sequences of repeating events using the star operator. The following example shows how the star and the negation constructs can be combined to express the fact that a particular event does not happen within a specific time interval.

EXAMPLE 6: [Star Sequence & Negation] 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 > 100,000),
    * ! 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 > 100,000)
  )
  -> write("Good Customer: %s \n", X.Owner).
```

□

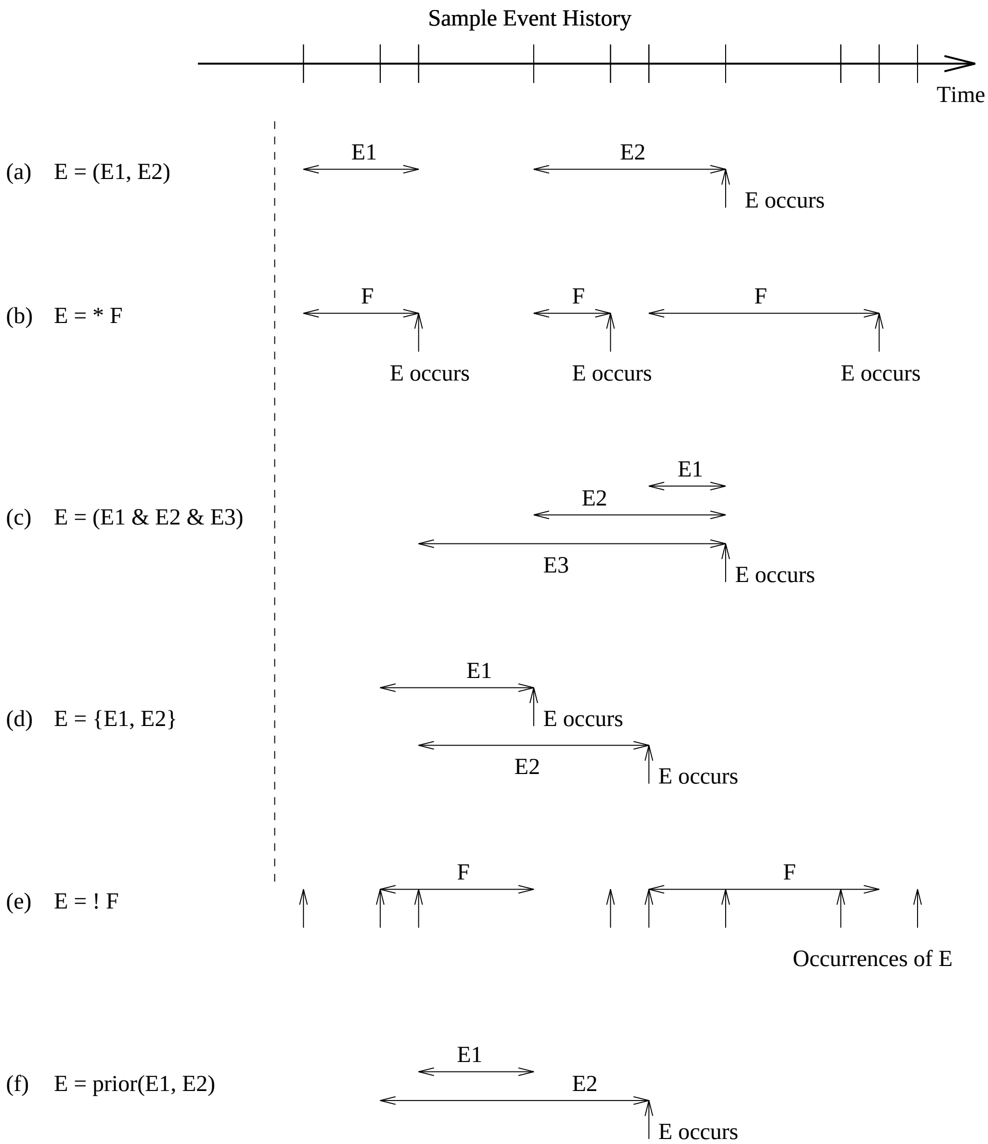

Figure 2. Examples of EPL composite events.

4. Semantics of EPL

In this section, we demonstrate our method for defining the semantics of composite event languages. After introducing the basic ideas, we define the formal semantics of EPL rules.

4.1. Event Tables and Event Histories

In order to define the semantics of EPL expressions, we need to formally define the notion of *event histories*, against which these expressions are evaluated.

Our starting point is the *event tables* for the underlying relational database. According to the practice in most commercial databases, each *event table* corresponds to a particular *basic event type*, and accumulates the *time-stamped* occurrences of this event type. A wide variety of events can be monitored in this fashion, including events other than database updates, such as read and write actions, or application-specific external events. Thus, our method can be applied to systems that support different kinds of events. Here, however, we only consider database modification events (i.e., inserts, deletes and updates), which prevail in active relational databases. For instance, insertions into the ACC table are recorded in the following event table:

```
ins_ACC(Accno, Owner, Type, Balance, Timestamp).
```

The `del_ACC` table has a similar format, while for updates of `ACC`, both the old and new values are recorded in event table `upd_ACC`, as follows:

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

Furthermore, each module maintains a log of the occurrences of the basic event types it monitors. This log can be stored as a relation, as it is shown below for the case of an example module `Mod`:

```
log_Mod(EvtType, Timestamp).
```

The second step is to define module-specific histories and event relations. This is necessary, since an EPL expression is evaluated with respect to a particular module. For the sake of simplicity, we assume a single module of discourse `Mod`, however, all the following definitions can be replicated or generalized to handle several modules.

A sample event history for our module follows:

EXAMPLE 7: A sample event history for module `Mod`. The timestamp of the first entry corresponds to the time the module is activated.

Table 1. A sample event history for a module.

hist_Mod	EvtType	Timestamp	Stage
	—	1400	0
	upd_ACC	1423	1
	upd_ACC	1425	2
	ins_ACC	1430	3
	ins_ACC	1502	4

□

A new column called *stage* has been introduced to hold the sequence numbers of the events. The stage values define a total order on the distinct timestamps, *modulo the fact that simultaneous events are given the same timestamp.* This allows the expression of composite event properties that are based on the *temporal order* of occurrence of their component simple events. Conditions based on explicit time can also be expressed using the timestamps of simple events.

These observations lead naturally to the use of $Datalog_{1S}$. In fact, the history relation `hist_Mod` could be defined by the following $Datalog_{1S}$ rules, where the temporal argument corresponds to the *stage* parameter.[2]

```
hist_Mod("−", 1400, 0).
hist_Mod(EvtType,T2,s(J)) ←  hist_Mod(_,T1,J),
                              log_Mod(EvtType,T2),
                              T2 > T1,  ¬between(T1,T2).

between(T1,T2) ←              log_Mod(EvtType,T),
                              T1 < T,  T < T2.
```

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 this stage, timestamped with the transaction's commit time. Basic events that share a stage number are considered to be *simultaneous* and are further discussed in Section 4.6.

Finally, for each *basic event type* monitored by our module, a predicate that describes the history of the event's occurrences within this module can be defined. For each occurrence of this event type, the predicate contains a tuple with the attribute bindings and the stage of the occurrence. For instance, for the basic event type `ins_owns`, we have:

```
ins_ACC_Mod(Accno,Owner,Type,Bal,Timestamp,J) ←
                                hist_Mod(ins_ACC,Timestamp,J),
                                ins_ACC(Accno,Owner,Type,Bal,Timestamp).
```

4.2. Event Satisfaction

To formally define the meaning of arbitrary EPL event expressions, we introduce the notion of *satisfaction predicates*. For a particular *event type*, simple or composite, its *satisfaction predicate* describes the history of the occurrences of this event type.

We start with qualified basic events. According to the notation used so far, a qualified basic event is represented as `E = evtkind(R(X), q(X))`, where **evtkind** is **ins**, **del** or **upd**, `R` is a relation name, and `q` denotes the event's condition expression, which can refer to the attribute values of tuple variable `X`.[3] The rule template for the *satisfaction predicate* of such an event is:

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

The predicate **evtkind_R**(X, J) is defined as explained in the end of the previous Section. For instance, within the context of a particular module, the satisfaction predicate for event `E = ins(ACC(X), X.Type = "Savings")` is defined as follows:

$$\begin{aligned}\mathtt{sat_E(Accno,Owner,Type,Bal,Timestamp,J)} \leftarrow \\ \mathtt{ins\_ACC_{Mod}(Accno,Owner,Type,Bal,Timestamp,J)}, \\ \mathtt{Type} = \text{``}\mathtt{Savings}\text{''}.\end{aligned}$$

The concept of "an event *immediately following* another event" can also be expressed. Take, for instance, the following example, where `E` represents the immediate sequence of Example 3:

EXAMPLE 8: The immediate sequence

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

Its semantics is defined by the following three $Datalog_{1S}$ rules:

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

□

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 `E` coincides with that of $\mathtt{sat_2}$.

In general, EPL expressions can be naturally mapped to $Datalog_{1S}$ rules. In this thesis, we have developed a procedure, where each EPL event expressions is translated to an equivalent set of $Datalog_{1S}$ rules. The resulting set of rules has a well-established formal semantics (model-theoretic and fixpoint-based) [3]. To formalize the translation, we represent EPL event expressions by their parse trees, using the following prefix notation:

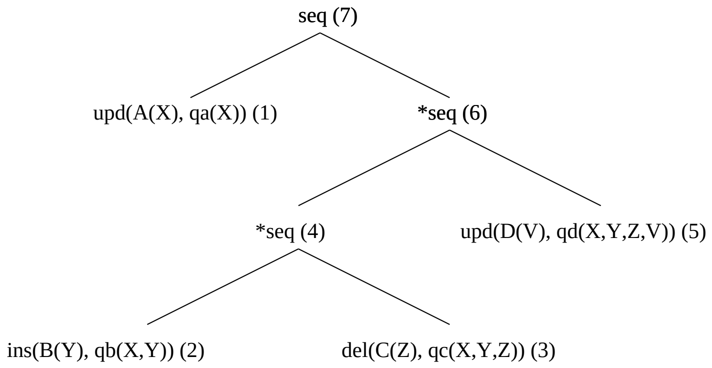

Figure 3. The parse tree for Example 9.

1. $seq(E_i, E_j) \equiv (E_i, E_j)$.[4]
2. $*seq(E_i, E_j) \equiv (*E_i, E_j)$.[5]
3. $and(E_i, E_j) \equiv E_i \ \& \ E_j$.
4. $or(E_i, E_j) \equiv \{E_i, E_j\}$.
5. $neg(E) \equiv\ !E$.
6. $\varepsilon \equiv$ *no event*.

EXAMPLE 9: An EPL expression with Immediate and Star Sequences.

$$\begin{array}{l}\mathtt{E} = (\ \mathtt{upd(A(X),q_a(X))},\\ \qquad *\ (*\ \mathtt{ins(B(Y),q_b(X,Y))},\ \ \mathtt{del(C(Z),q_c(X,Y,Z))}\ \ ),\\ \qquad \mathtt{upd(D(V),q_d(X,Y,Z,V))}\ \ )\end{array}$$ □

The parse tree for the expression of Example 9 is shown in Figure 3. The nodes of the tree are numbered according to the postorder traversal sequence.

A node numbered **i** corresponds to a subevent; let us call this subevent $\mathtt{E_i}$. For instance, node **4** corresponds to subevent
$\mathtt{E_4} = *\mathtt{seq(ins(B(Y),q_b(X,Y)),\ \ del(C(Z),q_c(X,Y,Z)))}$.

The *satisfaction predicate* of $\mathtt{E_i}$ is denoted as $\mathtt{sat_i}$. Each tuple of $\mathtt{sat_i}$ contains the variable bindings for an instance of $\mathtt{E_i}$. In particular, at each stage in the event history, $\mathtt{sat_i}$ contains one tuple for each instance of $\mathtt{E_i}$ that can still contribute towards the recognition of an instance of the root event **E**.

4.3. The Translation Method

As demonstrated in Example 8, the $Datalog_{1S}$ rules derived from the translation of a composite EPL expression must model (i) the temporal precedences among its various subevents, and (ii) the transmission of variable bindings according to the scope rules of the language.

Table 2 describes how this information is derived for each basic EPL construct. Formally, it defines a simple attribute grammar [20, 11].

Table 2. An attribute grammar for translating EPL expressions to $Datalog_{1S}$ rules.

EvtType E	PPS	EVar(E)	IVar
evt(R(X))	—	{X}	—
seq(F,G)	PPS(F) = PPS(E) PPS(G) = {F}	EVar(F) ∪ EVar(G)	IVar(F) = IVar(E) IVar(G) = IVar(E) ∪ EVar(F)
*seq(F,G)	PPS(F) = {F} ∪ PPS(E) PPS(G) = {F}	EVar(F) ∪ EVar(G)	IVar(F) = IVar(E) IVar(G) = IVar(E) ∪ EVar(F)
or(F,G)	PPS(F) = PPS(E) PPS(G) = PPS(E)	∅	IVar(F) = IVar(E) IVar(G) = IVar(E)
and(F,G)	PPS(F) = PPS(E) PPS(G) = PPS(E)	EVar(F) ∪ EVar(G)	IVar(F) = IVar(E) IVar(G) = IVar(E)
neg(F)	PPS(F) = PPS(E)	∅	IVar(F) = IVar(E)

The first column of Table 2 lists the various EPL event types. For each subevent `Q` of a EPL event `E`, the second column in the Table defines the *Possible Predecessors Set* of `Q`, denoted as `PPS(Q)`.

Definition 2. Assuming an event expression `E`, and two subevents `P` and `Q` of `E`, then we say that *"P is a possible predecessor of Q"*, if within an instance of `E`, an instance of `Q` may begin *immediately after* (at the next stage after) the end of an instance of `P`. Because of the star and the disjunction operators, a particular subevent may have many possible predecessors.

As an example, consider the immediate sequence construct `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 2 describe the scope rules for variables in EPL. For each event expression `E`, the third column specifies the set of *exported variables* from `E`. These are variables defined in `E`—i.e., variables whose first appearance is in basic events within this expression—whose scopes extend past `E`. Thus, exported variables from `E` can be used in the condition expressions of following events. The fourth column defines for each subevent `Q` of `E`, the set of variables *imported* into `Q`. These are variables defined before `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, some variables from previous events might be imported into `E`, 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`.

Using this attribute grammar, the generation of the actual rules is simple, as shown in Table 3.

Table 3. $Datalog_{1S}$ rule templates for the basic constructs of EPL.

Event Type E	$Datalog_{1S}$ Rule Templates
Any Basic Event	$any(J) \leftarrow hist\_Mod(\_,\_,J).$
Simple Event evt(R(X), Cond)	If PPS(E) $= \emptyset$, $sat_E(X,J) \leftarrow evt\_R(X,J),\ Cond(X).$ If PPS(E) $\neq \emptyset$, for each P $\in$ PPS(E) $sat_E(IV,X,s(J)) \leftarrow evt\_R(X,s(J)),\ sat_P(IV,\_,J),$ $Cond(IV,X).$
Condition-Only Goal Cond	for each P $\in$ PPS(E), $sat_E(IV,J) \leftarrow sat_P(IV,\_,J),\ Cond(IV).$
seq(F(X),G(Y))	$sat_E(IV,X,Y,J) \leftarrow sat_G(IV,X,Y,J).$
*seq(F(X),G(Y))	$sat_E(IV,X,Y,J) \leftarrow sat_G(IV,X,Y,J).$
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))	If PPS(E) $= \emptyset$, $sat_E(J) \leftarrow any(J),\ \neg sat_F(\_,J).$ If PPS(E) $\neq \emptyset$, 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	If PPS(E) $\neq \emptyset$, for each P $\in$ PPS(E), $sat_E(IV,J) \leftarrow sat_P(IV,\_,J)$

The first row in this Table defines directly the event `any` (see Section 3.2), using the event history relation for the module of discourse `Mod`. This event is used in several of the following examples.

The second row deals with *simple events*. If a simple event `E` has no possible predecessors, a single rule is produced, as explained in the beginning of Section 4.2. If it has some possible predecessors, then $\mathtt{sat_E}$ is satisfied at stage `s(J)`, when: (1) a possible predecessor of `E` is satisfied at stage `J`, (2) `E` occurs at stage `s(J)`, and (3) the condition of `E` is satisfied. Thus, for each one of the possible predecessors of `E`, one rule that defines these properties is produced. Note that in the goal $\mathtt{sat_P(IV,\_,J)}$ for a predecessor `P` of `E`, `IV` denotes the *set* of exported variables from `P`, and the anonymous variable "_" denotes all variables that are kept local in `P`.

The rules for *condition-only goals* are similar, except for the fact that such a goal must necessarily have some possible predecessors.

For each of the remaining event types, `X` and `Y` denote *sets* of exported variables defined in the subevents of this event types. The variables of the satisfaction predicate for a composite event type `E` consist in the union of the variables exported from `E`, plus the variables imported into it (denoted as `IV`).

In the case of an immediate sequence or a star sequence expression, an extra *copy-rule* is added after the rules for the components of this expression are generated. The copy-rule (shown in the fourth and fifth rows of Table 3) expresses the fact that the sequence is satisfied when the last component event is satisfied—see, for instance, the third rule in Example 8.

The rules for disjunction and conjunction are also straightforward. Observe that in a conjunction, all the variables defined in its conjuncts are exported, whereas in a disjunction, none of the variables defined in its disjuncts is exported. The rule for *negated events* is explained in detail in Section 4.7.

In the remainder of this Section, some of the most interesting points regarding various EPL constructs are further elaborated. Several examples are presented.

4.4. Star Sequences

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

Consider e.g., the EPL expression `E(X,Y,Z) = ( F(X), * G(Y), H(Z) )`, which specifies an immediate sequence consisting of an instance of `F`, followed by one or more instances of `G`, followed finally by an instance of `H`. According to Table 2, `PPS(G) =` {`F, G`} and `PPS(H) =` {`G`}. The variable `Y` exported from `G`, which can be used in the condition of `H` refers to the last occurrence of `G` in the star sequence. In addition to that, exported variables from `G` may include aggregate columns for the temporal aggregates defined on this star sequence. The semantics of temporal aggregates are clarified in the next section.

Consider now the more complicated case of Example 9, where star expressions are nested. Referring to Figure 3 and using Table 2, we get:

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

Using the *PPS* sets of the basic events, as well as the scope rules of EPL, the following $Datalog_{1S}$ rules are derived for Example 9:

EXAMPLE 10: The $Datalog_{1S}$ semantics for Example 9.

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

□

Consider, for instance, the basic event $\mathtt{E_2 = ins(B(Y),q_b(X,Y))}$ in Figure 3. This basic event may immediately follow either an occurrence of basic event $\mathtt{E_1}$, or another occurrence of $\mathtt{E_2}$ (because of the innermost star), or an occurrence of a star subsequence $\mathtt{E_4}$ (because of the outermost star).

The satisfaction predicates and the corresponding *copy-rules* for the `seq` and `*seq` nodes could be eliminated, resulting in a smaller set of rules. Nevertheless, 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 the simple events within the scope of these variables.

4.5. Relaxed Sequences

In section 3.2, some 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 for these additional constructs. Yet, a direct translation is often desirable, as it leads to much more efficient implementation. For instance, the event `any` does need not be defined as the disjunction of all basic events in the module of interest, but can be simply defined as it is shown in the first row of Table 3.

The same reasoning applies to relaxed sequences, which can be treated similarly to immediate sequences. In particular, the rules of Table 2 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` can begin at any later stage, rather than immediately after an occurrence of `F`. This semantics can be modeled by using an *auxiliary predicate* $\mathtt{has\_sat_1}$, as it is illustrated in the following Example.

EXAMPLE 11: A two-event relaxed sequence:

$$\mathtt{E = [\ upd(ACC(X),q_1(X)),\ \ upd(ACC(Y),q_2(X,Y))\ ],}$$

and its translation:

$$\begin{array}{ll}
\mathtt{sat_1(X,J)} \leftarrow & \mathtt{upd_ACC(X,J),\ q_1(X).} \\
\mathtt{has_sat_1(X,J)} \leftarrow & \mathtt{sat_1(X,J).} \\
\mathtt{has_sat_1(X,s(J))} \leftarrow & \mathtt{any(s(J)),\ has_sat_1(X,J).} \\
\mathtt{sat_2(X,Y,s(J))} \leftarrow & \mathtt{upd_ACC(Y,s(J)),\ has_sat_1(X,J),\ q_2(X,Y).}
\end{array}$$

□

For each stage `J` in the event history, the auxiliary predicate $\mathtt{has\_sat_1(X,J)}$ contains one tuple for each occurrence of $\mathtt{upd(ACC(X),q_1(X))}$ that happened *before or at this stage*. Then, an occurrence of $\mathtt{upd(ACC(Y),q_2(X,Y))}$ is matched with all previous occurrences of $\mathtt{upd(ACC(X),q_1(X))}$, as expressed by the rule for $\mathtt{sat_2}$.

4.6. Conjunction and Simultaneous Events

The simplest use of conjunction is for expressing *simultaneous* simple events. Consider, for instance, the following example, where composite event `E` occurs when the first basic event is immediately followed by the simultaneous occurrence of the last two basic events:

EXAMPLE 12: A composite event with simultaneous basic events:

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

Its translation into $Datalog_{1S}$ rules:

$$\begin{array}{ll}
\mathtt{sat_1(X,J)} \leftarrow & \mathtt{upd_A(X,J).} \\
\mathtt{sat_2(X,Y,J)} \leftarrow & \mathtt{ins_B(Y,s(J)),\ sat_1(X,J).} \\
\mathtt{sat_3(X,Z,J)} \leftarrow & \mathtt{del_C(Z,s(J)),\ sat_1(X,J).} \\
\mathtt{sat_E(X,Y,Z,J)} \leftarrow & \mathtt{sat_2(X,Y,J),\ sat_3(X,Z,J).}
\end{array}$$

□

For simple events, which are instantaneous, the meaning of temporal notions such as "simultaneous" or "prior to" is well-established by their use in everyday life, and the semantics of EPL operators conforms to this intuitive meaning. However, a composite event instance has duration, which is the time interval between the first and last simple events that are part of this instance. Thus, in the case of composite events, the meaning of concepts such as "prior" and "simultaneous" can be ambiguous in natural languages. Two composite events might be called simultaneous if (i) they are completed at the same time, or (ii) they start at the same time, or (iii) they start and complete at the same time, or (iv) they overlap in time.

We have adopted the first interpretation, which is based on *event completion*. As shown in Figure 2, 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).

This choice seems to be the most practical one in database applications, and it also dovetails with the "immediately after" activation semantics, used in EPL. Under this rule execution scheme, the action in the head of a rule is fired as soon as the composite event in its body occurs. For instance, consider the following EPL rule, where B_1, B_2 and B_3 denote basic events:

$$[B_1, B_2] \ \& \ B_3 \ \rightarrow \ A$$

According to the semantics of the conjunction operator, this composite event occurs at the time granule in which both B_2 and B_3 occur, provided that B_1 has occurred before. The action A can be fired immediately after that.

Observe that if we were to apply interpretation (ii) on this rule, this would mean that the composite event $[B_1, B_2]$ and the event B_3 must both start at the same time points. However, the satisfaction of this condition would not allow A to take place at once; the rule can only be triggered after B_2 also occurs at some time later, yielding a behavior that does not dovetail with the "immediately after" activation semantics. A similar reasoning applies to the other interpretations mentioned above. Nevertheless, these alternative patterns of concurrency can be expressed explicitly in EPL. For instance, the pattern corresponding to interpretation (ii) can be simply expressed as:

$$[(B_1 \& B_3), \ B_2] \ \rightarrow \ A$$

One very important use of the conjunction construct is for expressing sequences based on event occurrences, as opposed to sequences of event instances—see Section 3.2.1 for the definition of these terms. An example is the definition of `prior`, which is repeated here:

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

Assuming that (a) the rules for F(X) and G(Y) have been generated and (b) an auxiliary predicate $\mathtt{has\_sat_F}$ has been defined according to the way shown in Example 11 for $\mathtt{has\_sat_1}$, the satisfaction predicate for E is defined as:

$$\mathtt{sat_E(X,Y,s(J))} \leftarrow \ \mathtt{sat_G(Y,s(J))}, \ \mathtt{has\_sat_F(X,J)}.$$

In general, the set of rules generated for a conjunctive event consists of the union of the sets of rules generated for its conjuncts, plus one extra rule. Thus, we do not have an exponential blow-up problem.

4.7. Negation

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

Using $Datalog_{1S}$, the semantics of general negation can be easily defined. The following example illustrates the case of a negated simple event (domain variables are used):

EXAMPLE 13: Translation for `E =!ins(ACC(X),X.Type = "Checking")`.

$$\begin{array}{ll} \mathtt{sat_F(Accno, Owner, Type, Balance, Timestamp, J)} \leftarrow & \\ & \mathtt{ins\_ACC(Accno, Owner, Type, Balance, Timestamp, J)}, \\ & \mathtt{Type} = \text{“}\mathtt{Checking}\text{”}. \\ \mathtt{sat_E(J)} \leftarrow & \mathtt{any(J)},\ \neg \mathtt{sat_F(\_,\_,\_,\_,\_,J)}. \end{array}$$

□

The first rule defines the satisfaction predicate for the simple event `F = ins(ACC(X), X.Type = ''Checking'')`. The second rule expresses the fact that `E` occurs at every stage where *some* basic event occurs, but `F` does not occur. Thus, with respect to the event history of a module, `E` occurs at every stage in this history, except when a creation of a "Checking" account is recorded (occurs).

As these rules show, the variables defined inside a negated event are not exported outside the event. This restriction ensures the *safety* and *domain-independence* of EPL expressions [21, 33].

The general case is similar. The second rule of Example 13 can still be used for the negated event `E = !F`, in the case where `F` is a composite 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*.

Example 14 and Figure 4 further illustrate how negated composite events are handled. This example contains also a disjunctive event.

EXAMPLE 14: A EPL expression with a negated disjunction.

$$\mathtt{(\ upd(A(X)),\ !\ \{\ upd(B(Y),q_b(X,Y)),\ \ ins(C(Z),q_c(X,Z))\ \}\ )}$$

□

The satisfaction of negated event $\mathtt{E_5}$ (at a stage where neither $\mathtt{E_2}$, nor $\mathtt{E_3}$ occur) must follow the occurrence of basic event $\mathtt{E_1}$.

Since a negated event instance has *single stage duration*, negated events are treated similarly to basic events, as far as temporal ordering is concerned. Specifically, one rule of the form shown in the 8-th row of Table 3 is created for each of the possible predecessors of a negated event. In this example, the only possible predecessor of the negated event $\mathtt{E_5}$ is $\mathtt{E_1}$. The $Datalog_{1S}$ rules for Example 14 follow:

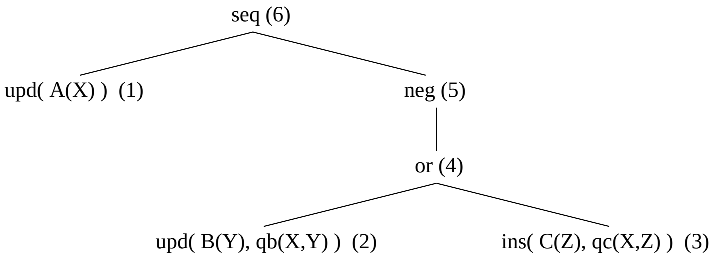

Figure 4. The parse tree for Example 14.

$$
\begin{array}{ll}
\mathtt{sat}_1(\mathtt{X},\mathtt{J}) \leftarrow & \mathtt{upd\_A}(\mathtt{X},\mathtt{J}). \\
\mathtt{sat}_2(\mathtt{X},\mathtt{Y},\mathtt{s}(\mathtt{J})) \leftarrow & \mathtt{upd\_B}(\mathtt{Y},\mathtt{s}(\mathtt{J})),\ \ \mathtt{q}_\mathtt{b}(\mathtt{X},\mathtt{Y}). \\
\mathtt{sat}_3(\mathtt{X},\mathtt{Z},\mathtt{s}(\mathtt{J})) \leftarrow & \mathtt{ins\_C}(\mathtt{Z},\mathtt{s}(\mathtt{J})),\ \ \mathtt{q}_\mathtt{c}(\mathtt{X},\mathtt{Z}). \\
\mathtt{sat}_4(\mathtt{X},\mathtt{J}) \leftarrow & \mathtt{sat}_2(\mathtt{X},\mathtt{Y},\mathtt{J}). \\
\mathtt{sat}_4(\mathtt{X},\mathtt{J}) \leftarrow & \mathtt{sat}_3(\mathtt{X},\mathtt{Z},\mathtt{J}). \\
\mathtt{sat}_5(\mathtt{X},\mathtt{s}(\mathtt{J})) \leftarrow & \mathtt{any}(\mathtt{s}(\mathtt{J})),\ \ \mathtt{sat}_1(\mathtt{X},\mathtt{J}),\ \ \neg\mathtt{sat}_4(\_,\mathtt{s}(\mathtt{J})). \\
\mathtt{sat}_\mathtt{E}(\mathtt{X},\mathtt{J}) \leftarrow & \mathtt{sat}_5(\mathtt{X},\mathtt{J}).
\end{array}
$$

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

$\mathtt{E}_1$ is also considered to be a possible predecessor of $\mathtt{E}_2$ and $\mathtt{E}_3$. This is because 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`. Thus, using Table 2, we get, for instance:

$$\mathtt{PPS}(2) = \mathtt{PPS}(4) = \mathtt{PPS}(5) = \{1\}$$

Finally, observe that while variables defined in a negated event are not exported outside it, variables defined in preceding events can be used inside the negated event. For instance, in Example 14, variable `X` can be used in the conditions $\mathtt{q}_\mathtt{b}$ and $\mathtt{q}_\mathtt{c}$ (see also Example 6). This design decision is consistent with the use of negation in deductive databases, where the safety restriction is relaxed to allow the use of *ground variables* in negated goals.

4.8. Semantics of EPL Rules

So far, we have defined the meaning of event expressions that appear in the bodies of EPL rules. Our approach can be easily extended to define the semantics of rule actions, as sketched here.

Let us assume that the event history relation for a module, also includes the actions of the EPL rules in this module. The actions are timestamped, in the same way basic events are.

Let `r` be an EPL rule with body (event) `E(X)` and head (action) `A(X)`. Also, let `H` be an event history. A *satisfaction instance* for the body of `r` is a pair $(\mathtt{x},\mathtt{t})_{\mathtt{H}}$, such that `E(x)` occurs at time `t` in the given history `H`. We say that the rule `r` is satisfied for $(\mathtt{x},\mathtt{t})_{\mathtt{H}}$, if `H` contains an action `A(x)` with timestamp $\mathtt{t_A} > \mathtt{t}$. Equivalently, this action must occur at a *stage later than the stage* where `E(x)` occurs. Finally, `r` is satisfied by `H`, if `r` is satisfied for every satisfaction instance of the body of `r`.

Thus, we have reduced the notion of correctness of EPL rules to the notion of satisfaction of certain $Datalog_{1S}$ predicates, which are evaluated against an event history. Essentially, we require that once the body of a rule becomes satisfied, the head action is later fired. Of course, this requirement can be made more stringent with liveness, or real-time conditions, by prescribing for instance, a maximum delay between the satisfaction of a rule's body and the timestamp of the action in the rule's head.

In this framework, actions that have duration can be modeled as two events — *action-start* and *action-end*. Likewise, a continuous event in the body of a rule can be treated as a composite event, characterized by its start and completion times.

5. From Semantics to Efficient Implementation

The formal semantics framework previously described is conducive to efficient implementation. Here, we outline two different implementation approaches, that facilitate the computation of the produced $Datalog_{1S}$ rules in an *incremental and history-less* manner.

The first approach sees the detection of a composite event as proceeding through a sequence of states as in a Finite State Automaton. In this regard, $Datalog_{1S}$ rules can be viewed as defining transitions between two states. Every time a basic event occurs, a new stage is added to the system—simultaneous basic events create a single new stage. At every stage, the *state* of all satisfaction predicates is updated by using (a) the state of the satisfaction predicates in the previous stage, and (b) the new basic event(s).

Thus, $Datalog_{1S}$ rules can be implemented using event tables and state tables. In general, $\mathtt{sat_E}(\mathtt{X},\mathtt{J})$ can be viewed as denoting the *state* of satisfaction predicate $\mathtt{sat_E}(\mathtt{X})$, at stage `J`. Consider, for instance, the second rule in Example 10:

$$\mathtt{sat_2(X,Y,s(J))} \leftarrow \mathtt{ins\_B(Y,s(J)),\ sat_1(X,J),\ q_b(X,Y).}$$

The predicate `ins_B(Y, J)` represents the occurrences of event type `ins_B` in the event history input stream. For each such occurrence, the condition $\mathtt{q_b(X,Y)}$ is evaluated using the state of $\mathtt{sat_1(X)}$ at the previous stage. For each new pair `(X, Y)` satisfying the body of the above rule, a new entry is added to the new state of $\mathtt{sat_2}$. When all basic events at the new stage in the history are serviced, we no longer need the state of the satisfaction predicates at the previous stage.

In summary, at each stage in the event history, the basic computation cycle consists of (1) servicing the new event entry(ies), (2) evaluating the conditions of rules, and (3) updating the state tables of the satisfaction predicates.

More complicated patterns such as disjunctions, conjunctions and negated events can also be implemented by using additional state tables and propagating changes to composite-event state tables as indicated by the corresponding $Datalog_{1S}$ rules. However, this approach requires a specialized implementation scheme for each construct supported by the language. Thus, while it can be highly optimized, it may also require significant development effort.

The second approach is probably simpler, and consists in implementing $Datalog_{1S}$ rules directly in an existing rule processor.

Using a general-purpose deductive database system, such as $\mathcal{LDL}$++ [2], the $Datalog_{1S}$ rules can be translated almost verbatim, but for an efficient implementation, the compiler must take advantage of the particular syntactic structure of these rules, so that their execution is optimized. Fortunately, $Datalog_{1S}$ programs produced by our translation procedure satisfy a particular form of local stratification, called *XY–stratification* [35, 36]. This is because the stage values in the bodies of these rules are either the same as those in their heads (X-rules) or one greater (Y-rules). As shown in the next Section, X-rules and Y-rules are especially important in the presence of certain negated goals.

Generally, programs with X-rules and Y-rules can be executed by memorizing only the current and past values of predicates, instead of all the past values, as normally required by fixpoint computations. Thus, the implementation of such programs can be optimized, by maintaining only two instances of predicate tables during execution, the new one and the old one. After each iteration of the fixpoint computation, the two copies of the predicate tables exchange their roles, one holding the old tuples, and the other receiving the new tuples.

Moreover, since the current stage value is the same for all rules being computed, this value can be kept globally, rather than with each tuple stored. This leads to further optimizations. Consider for instance the translation of the relaxed sequence in Example 11:

$$
\begin{array}{ll}
\mathtt{sat_1(X,J)} \leftarrow & \mathtt{upd\_ACC(X,J),\ q_1(X).} \\
\mathtt{has\_sat_1(X,J)} \leftarrow & \mathtt{sat_1(X,J).} \\
\mathtt{has\_sat_1(X,s(J))} \leftarrow & \mathtt{any(s(J)),\ has\_sat_1(X,J).} \\
\mathtt{sat_2(X,Y,s(J))} \leftarrow & \mathtt{upd\_ACC(Y,s(J)),\ has\_sat_1(X,J),\ q_2(X,Y).}
\end{array}
$$

Observe that the third rule is essentially a *copy* rule, that simply copies values from the old state to the new one, at the occurrence of any basic event. The fact that the stage value is handled separately means that the stage argument can be dropped from the predicates, and the new relation for $\mathtt{has\_sat_1}$ can be simply constructed by taking the old one as is. A smart compiler should recognize this situation and keep only one copy of this predicate, rather than an old copy and a new one [35]. Thus, the third rule above is ignored, while the second one adds to this copy.

To ensure efficient computation in the case of large rule sets, it is also important to speed-up the identification of firable rules, by building an index that for each basic event type points to the rules that might be fired by occurrences of this event type. Using $\mathcal{LDL}$++, these indexes need to be supported through user-defined tables. Another alternative is to use a specialized *forward-chaining* system, e.g., CLIPS [26]. Such systems provide a sound basis for implementation, due to their efficient rule activation mechanisms. Typically, they also utilize specialized internal event indexing mechanisms, which are very effective in identifying relevant rules for each event occurrence, and firing these rules immediately.

As per the discussion in the next Sections, the implementation techniques presented here are not limited to EPL; rather, they are applicable to other composite event languages as well.

6. Application to Other Systems

A very significant advantage of the proposed method is its generality, whereby it can be employed for the formal definition of constructs and concepts proposed in previous systems. This is demonstrated in this Section.

6.1. ODE

The case of ODE is straightforward, given the many similarities between the EPL and ODE constructs. The most important difference is that, in ODE, the *relaxed sequence* is a basic construct and the *immediate sequence* is a derived one. Thus, the semantics of ODE can be defined by a similar syntax-directed translation.

In fact, the application of the technique to ODE clarifies the definition of disjunction, which seems to be problematic in [15]. There, only conjunction is defined as a primitive construct, while disjunction is defined through negation, via De Morgan's law. However, this approach leads to technical difficulties, for the following two reasons: (i) as explained in section 4.7, a negated event has always single stage duration, whereas, an instance of a disjunction may extend over an interval in time, and (ii) A parameterized disjunctive event cannot be defined through negation, because of the *safety* problem.

6.2. The SNOOP Language

The approach used here to define the semantics of EPL can also be applied to other languages, including SNOOP and the SAMOS' language that present substantial differences with respect to EPL and ODE. For instance, at the language level, one of the differences is that in SNOOP and SAMOS, the meaning of the sequence of two events $\mathtt{E_1}$ and $\mathtt{E_2}$ is equivalent to the meaning of $\mathtt{prior(E_1,E_2)}$ in EPL and ODE. From an implementation standpoint, the differences are even more important, since SAMOS and SNOOP have used Petri Nets and Event Graphs for their implementation.

In the next section, we define the semantics of the main SNOOP constructs and we explain how its concept of *parameter contexts* can be incorporated in our method. Then, we give a very detailed example of how we can define the execution semantics of Petri Nets, used in the definition and implementation of SAMOS.

In this Section, we define the semantics of the main SNOOP constructs and we explain how its concept of *parameter contexts* can be incorporated in our method.

6.2.1. Main Constructs.

1. **SEQ (;).**
   Since the event $\mathtt{E} = \mathtt{E_1(X);\ E_2(Y)}$ is equivalent to $\mathtt{prior(E_1(X),\ E_2(Y))}$ in EPL and ODE, the rule for $\mathtt{sat_E}$ is defined as it is explained in Sections 4.5 and 4.6:

   $$\mathtt{sat_E(X, Y, s(J))} \leftarrow \mathtt{sat_{E_2}(Y, s(J)),\ has\_sat_{E_1}(X, J).}$$

2. **OR.** This is similar to EPL's disjunction, but the scope rules are different. In SNOOP, the parameters of a disjunction's instance are the parameters of the disjunct that actually occurs. This is different from EPL, where the variables inside a disjunction are not exported outside it.

3. **Aperiodic Operator A.** The event $\mathtt{E} = \mathtt{A(E_1, E_2, E_3)}$ occurs each time $\mathtt{E_2}$ occurs, within the *closed* time interval starting with an occurrence of $\mathtt{E_1}$ and ending with the *first subsequent* occurrence of $\mathtt{E_3}$. The parameters of $\mathtt{E}$ are the parameters of $\mathtt{E_2}$. The translation into $Datalog_{1S}$ rules follows. It is assumed that the satisfaction predicates for $\mathtt{E_1, E_2}$ and $\mathtt{E_3}$ ($\mathtt{sat_1, sat_2}$ and $\mathtt{sat_3}$, respectively) have been separately defined.

   $$\begin{array}{ll} \mathtt{between_{1,3}(J)} \leftarrow & \mathtt{sat_1(\_, J).} \\ \mathtt{between_{1,3}(s(J))} \leftarrow & \mathtt{any(s(J)),\ between_{1,3}(J),\ \neg sat_3(\_, J).} \\ \mathtt{sat_E(X, J)} \leftarrow & \mathtt{between_{1,3}(J),\ sat_2(X, J).} \end{array}$$

   The auxiliary predicate $\mathtt{between_{1,3}}$, is *true* at all stages in the closed time interval between an occurrence of $\mathtt{E_1}$ and the *first subsequent* occurrence of $\mathtt{E_3}$. $\mathtt{X}$ denotes the parameters of $\mathtt{E_2}$.

   For an outline of how to handle the aperiodic cumulative operator A^* (used mainly for computing aggregates), we refer to the discussion of the cumulative parameter context.

4. **NOT.** The event $\mathtt{E} = \neg\mathtt{(E_2)[E_1, E_3]}$ detects the non-occurrence of $\mathtt{E_2}$ in the *closed* interval between an occurrence of $\mathtt{E_1}$ and the *first subsequent* occurrence of $\mathtt{E_3}$. It is signaled on this occurrence of $\mathtt{E_3}$. The corresponding $Datalog_{1S}$ rules follow:

   $$\begin{array}{ll} \mathtt{between_{1,23}(J)} \leftarrow & \mathtt{sat_1(\_, J).} \\ \mathtt{between_{1,23}(s(J))} \leftarrow & \mathtt{any(s(J)),\ between_{1,23}(J),\ \neg sat_2(\_, J),\ \neg sat_3(\_, J).} \\ \mathtt{sat_E(J)} \leftarrow & \mathtt{between_{1,23}(J),\ sat_3(\_, J),\ \neg sat_2(\_, J).} \end{array}$$

The auxiliary predicate $\mathtt{between}_{1,23}$ is *true* at all stages in the closed time interval between an occurrence of $\mathtt{E}_1$ and the *first subsequent* occurrence of *either* $\mathtt{E}_2$ *or* $\mathtt{E}_3$.

The **AND** and **ANY** constructs of SNOOP have been implemented separately, for efficiency reasons. However, they can be expressed in terms of the **SEQ** and the **OR** constructs.

6.2.2. Parameter Contexts. A novelty offered by SNOOP is the concept of *parameters contexts* [5, 6]. This feature provides the ability to interpret temporal sequences of events in different ways, and thus, it facilitates the precise modeling of a wide range of applications.

Different parameter contexts arise from different interpretations of sequences in SNOOP and SAMOS, or from relaxed sequences in EPL and ODE. In this Section, we show that our method can be easily extended to handle the various parameter contexts of SNOOP. For the following SNOOP sequence example, we explain how its different interpretations can be modeled in $Datalog_{1S}$. These results can be generalized in a straightforward manner.

EXAMPLE 15: The SNOOP expression `E(X,Y) = A(X) ; B(Y))`, and the event history: $\ldots, A(1), \ldots, A(2), \ldots, B(1), \ldots, B(2)$,
where `A` and `B` denote basic events with parameters `X` and `Y` respectively. □

Depending on the parameter context of choice, different occurrences of $E(X,Y)$ will be recognized. For instance, $A(1)$ may be paired with each one of $B(1)$ and $B(2)$, or with $B(1)$ only. The following parameter contexts are supported by SNOOP:

- **Unrestricted Context**. In this parameter context, which has been assumed so far, all the instances of `E` are detected. These are:
  $(A(1); B(1))$, $(A(1); B(2))$, $(A(2); B(1))$ and $(A(2); B(2))$.

  According to the discussion of the SNOOP **SEQ** operator, the semantics of `E` in this context is defined as follows:

  $$\begin{array}{ll}
  \mathtt{sat}_1(\mathtt{X},\mathtt{J}) \leftarrow & \mathtt{A}(\mathtt{X},\mathtt{J}). \\
  \mathtt{has_sat}_1(\mathtt{X},\mathtt{J}) \leftarrow & \mathtt{sat}_1(\mathtt{X},\mathtt{J}). \\
  \mathtt{has_sat}_1(\mathtt{X},\mathtt{s}(\mathtt{J})) \leftarrow & \mathtt{any}(\mathtt{s}(\mathtt{J})),\ \mathtt{has_sat}_1(\mathtt{X},\mathtt{J}). \\
  & \\
  \mathtt{sat}_2(\mathtt{X},\mathtt{Y},\mathtt{s}(\mathtt{J})) \leftarrow & \mathtt{B}(\mathtt{Y},\mathtt{s}(\mathtt{J})),\ \mathtt{has_sat}_1(\mathtt{X},\mathtt{J}).
  \end{array}$$

  An *auxiliary predicate* $\mathtt{has\_sat}_1$ is used. To transform to different parameter contexts, we need to define other appropriate auxiliary predicates.

- **Recent Context.** In this context, the last occurrence of an event (simple or composite) subsumes the effect of all previous occurrences of the same event.

Thus, at each point in time, only the most recent occurrences of the various event types of interest are maintained. In our example, under the recent context, only the instances $(A(2); B(1))$ and $(A(2); B(2))$ would be recognized.

The following stratified $Datalog_{1S}$ rules define the semantics of this event expression, in the recent context:

$$\begin{array}{ll}
\mathtt{sat_1(X,J)} \leftarrow & \mathtt{A(X,J)}. \\
\mathtt{last_sat_1(X,J)} \leftarrow & \mathtt{sat_1(X,J)}. \\
\mathtt{last_sat_1(X,s(J))} \leftarrow & \mathtt{any(s(J)),\ last_sat_1(X,J),\ \neg sat_1(_,s(J))}. \\
\mathtt{sat_2(X,Y,s(J))} \leftarrow & \mathtt{B(Y,s(J)),\ last_sat_1(X,J)}.
\end{array}$$

For each stage `J` in the event history, the auxiliary predicate $\mathtt{last\_sat_1(X,J)}$ contains the parameter bindings for the *last* occurrence of `A(X)` that happened *before or at this stage*.

- **Chronicle Context**. In the chronicle context, when a composite event `E` occurs, its parameter bindings are obtained from the oldest *unused* occurrences of its component events that satisfy the temporal order requirements of `E`. This description implies that a particular simple event occurrence can participate in *at most one* instance of `E`.

  In our example, the first occurrence of A is paired with the first occurrence of B, resulting in the occurrence of $E(1,1) = (A(1); B(1))$. After that, $A(1)$ and $B(1)$ are discarded, since neither of them can be part of any other occurrence of E. Thus, when $B(2)$ occurs, it is paired with $A(2)$, which is the only *active* occurrence of A at this point, and $E(2,2) = (A(2); B(2))$ is generated.

  The following $Datalog_{1S}$ program provides this semantics:

$$\begin{array}{ll}
\mathtt{sat_1(X,J)} \leftarrow & \mathtt{A(X,J)}. \\
\mathtt{unused_1(X,J)} \leftarrow & \mathtt{sat_1(X,J)}. \\
\mathtt{unused_1(X,s(J))} \leftarrow & \mathtt{any(s(J)),\ unused_1(X,J),\ \neg sat_2(X,_,s(J))}. \\
\mathtt{not_oldest_1(X,J)} \leftarrow & \mathtt{unused_1(Y,J),\ Y.Time < X.Time}. \\
\mathtt{sat_2(X,Y,s(J))} \leftarrow & \mathtt{B(Y,s(J)),\ unused_1(X,J),\ \neg not_oldest_1(X,J)}.
\end{array}$$

  For each stage `J` in the event history, the auxiliary predicate $\mathtt{unused_1(X,J)}$ contains the parameter bindings for all occurrences of `A(X)` that happened *before or at this stage* and have not been matched with an occurrence of `B(Y)` yet. The rule for $\mathtt{sat_2}$ expresses the fact that an occurrence of `B(Y)` is matched with the *oldest unused* occurrence of `A(X)`, if any.

  Even though these rules are not stratified, they are locally stratified and *XY–stratified* [35]. Therefore, they can be evaluated in an incremental and efficient manner. To see that, observe that the computation of the contents of $\mathtt{unused_1}$

and $\texttt{not\_oldest}_1$ at stage J can be followed by the computation of the contents of $\texttt{sat}_2$ at stage s(J), which can then be followed by the computation of the contents of $\texttt{unused}_1$ 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.

- **Continuous Context.**

  Unlike the chronicle context, many instances of a composite event E may occur at the same time point in the continuous context. However, at this point, every basic event that participated in one of these instances is deleted from storage, and thus, it cannot be part of any future instance of E.

  In our example, the instances $(A(1); B(1))$ and $(A(2); B(1))$ are detected in the continuous context, when $B(1)$ occurs. At this point, $A(1), A(2)$ and $B(1)$ will be discarded, and thus, the occurrence of $B(2)$ will not produce any further occurrences of E.

  The following $Datalog_{1S}$ rules provide this semantics:

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

  Also, this set of rules is not stratified, however, it is *XY–stratified.*

- **Cumulative Context**. In this context, for each component event, all occurrences of the event are accumulated until the composite event is detected. Then, aggregation can be performed on these component event occurrences.

  In [25], we show that temporal aggregation in active rules with composite can be easily incorporated within our semantics framework.

6.3. SAMOS and Petri Nets

The most distinguishing feature of SAMOS is its event detection mechanism, which is based on *colored Petri Nets* [8]. A *colored Petri Net* is an extended version of a classical Petri Net [30] that allows the flow of parameter bindings through it. This provides a means for modeling the parameter passing within a composite event instance.

For each primitive composite event construct, an appropriate Petri Net structure has been defined [12]. The Petri Net for a complex event pattern can then be built by combining the Petri Net structures of its component events.

A *place* in a SAMOS Petri Net models an event type, basic or composite. It may contain one or more *tokens*, which correspond to distinct occurrences of this event

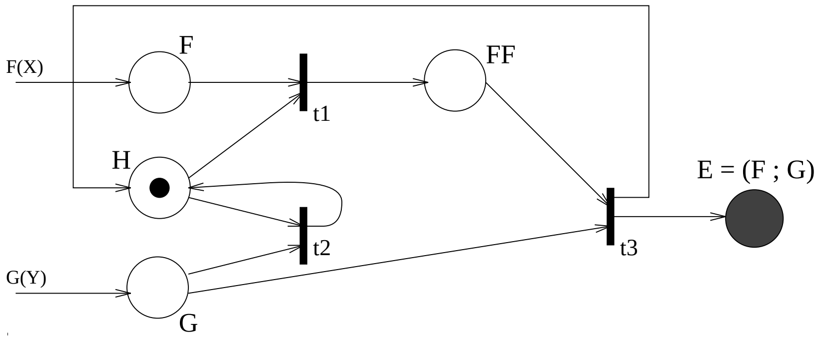

Figure 5. The Petri Net for the event pattern E = (F ; G).

type. The *token type* for a place specifies the variable signature of its event type. A place is *marked* if it contains at least one token, and multiple tokens are stored in a *FIFO queue*.

Event composition is accomplished by connecting places via *transitions*. A transition connects its input places (component event types) to its output place (composite event type). When all input places of a transition `t` are marked, the transition *fires* and from each input place, the token with the *oldest timestamp* (at the head of the FIFO queue) is removed. All these tokens are then matched to compose a new output token, which is deposited into `t`'s output place. The newly created token may in turn cause the firing of another transition, and so on. Thus, the arrival of a new token (event occurrence) in a Petri Net, could cause a sequence of transition firings.

For a detailed description of how Petri Nets can be used for the detection of composite events, the reader is referred to [14]. Here we illustrate the basic mechanism, by describing the behavior of the Petri Net in Figure 5, which is borrowed from [13]. This Petri Net corresponds to the *sequence event* `E(X,Y) = F(X) ; G(Y)`. The Figure shows the initial state of the Petri Net, where a token is contained in the *auxiliary place* `H`. Auxiliary places are used for modeling precedence dependencies among events.

This Petri Net accepts as input the occurrences of the simple event types `F(X)` and `G(Y)`. Every such occurrence corresponds to a new token inserted into the Petri Net; this token contains the parameter bindings of the event occurrence. Auxiliary place `H` is used to ensure that all occurrences of `G` are ignored, if there is no token in place `F`. Specifically, an occurrence of `G` at the state shown in the Figure will cause `t2` to fire and place `H` will be marked again. This has the effect of ignoring this occurrence of `G`.

However, an occurrence of `F` at this state will fire transition `t1` and the new token will mark place `FF`. The tokens of any later occurrences of `F` preceding the next occurrence of `G` will stay at place `F`. Different tokens of the same type are distinguished by their *"colors"*, i.e., their distinct variable bindings and timestamps.

When a token `G(y)` arrives while a token `F(x)` resides in place `FF`, the transition `t3` will fire. `F(x)` and `G(y)` will then be matched, and a new output token containing the parameter bindings (`x`, `y`) will be deposited into the output place `E`. Thus, a new occurrence of `E` will be signaled. At the same time, place `H` will be marked again, causing the *oldest* token among the ones in place `F` (if any) to move into place `FF`, waiting to be matched with the next occurrence of `G`.

From the above description, it follows that the event detection mechanism in SAMOS obeys the *chronicle* parameter context discussed in Section 6.2.2.

The execution semantics of our Petri Net is further illustrated by the example execution sequence in Table 4.

As it can be seen from this table, the occurrence of an external event (`F` or `G`) can cause a sequence of transition firings. Thus, the Petri Net proceeds through a sequence of intermediate *states*, until a *stable state* is reached, where no more transitions can fire.

Table 4. An execution sequence of the Petri Net in Figure 5.

Event	H	F	FF	G	E
	•	-	-	-	-
G(y_a)	•	-	-	y_a	-
t_2 fires	•	-	-	-	-
F(x_a)	•	x_a	-	-	-
t_1 fires	-	-	x_a	-	-
F(x_b)	-	x_b	x_a	-	-
G(y_b)	-	x_b	x_a	y_b	-
t_3 fires	•	x_b	-	-	(x_a,y_b)
t_1 fires	-	-	x_b	-	-
G(y_c)	-	-	x_b	y_c	-
t_3 fires	•	-	-	-	(x_b,y_c)

This sequence of states can naturally be defined in $Datalog_{1S}$, thus, capturing the execution semantics of SAMOS Petri Nets. A straightforward way to do that is by assigning a predicate to each component (place or transition) of a Petri Net. At each stage in the execution sequence, the contents of these predicates defines the state of the Petri Net. The dynamic behavior of the system is expressed via $Datalog_{1S}$ rules.

The $Datalog_{1S}$ program in Example 16 defines the execution semantics of the Petri Net in Figure 5. Here, the argument `J` defines the sequence of states this Petri Net goes through, as in Table 4, for instance. `F(X,J)` and `G(Y,J)` denote respectively, an occurrence of F or G at stage J. We assume that the occurrences of basic events are *non-simultaneous*, as required by Petri Nets.

A different predicate is defined for each place and each transition. Consider, for instance, the two rules for predicate in_F that corresponds to place F. The first rule says that the token for a new occurrence of F is deposited into the place of F at the stage of that occurrence. The second rule says that the token for a particular occurrence of F is contained in place F at stage s(J), if it is contained there at stage J and it is not removed by a firing of gate t1 at stage s(J).

EXAMPLE 16: The $Datalog_{1S}$ program for the Petri Net in Figure 5.

$$
\begin{array}{ll}
\mathtt{in\_F(X,J)} \leftarrow & \mathtt{F(X,J)}. \\
\mathtt{in\_F(X,s(J))} \leftarrow & \mathtt{any(s(J)),\ \ in\_F(X,J),\ \ \neg t_1(X,s(J))}. \\
\\
\mathtt{oldest\_in\_F(X,J)} \leftarrow & \mathtt{in\_F(X,J),\ \ \neg not\_oldest\_in\_F(X,J)}. \\
\mathtt{not\_oldest\_in\_F(X,J)} \leftarrow & \mathtt{in\_F(Y,J),\ \ Y.Timestamp < X.Timestamp}. \\
\\
\mathtt{in\_H(0)}. & \\
\mathtt{in\_H(s(J))} \leftarrow & \mathtt{any(s(J)),\ \ in\_H(J),\ \ \neg t_1(\_,s(J)),\ \ \neg t_2(\_,s(J))}. \\
\mathtt{in\_H(J)} \leftarrow & \mathtt{t_2(\_,J)}. \\
\mathtt{in\_H(J)} \leftarrow & \mathtt{t_3(\_,\_,J)}. \\
\\
\mathtt{in\_G(Y,J)} \leftarrow & \mathtt{G(Y,J)}. \\
\mathtt{in\_G(Y,s(J))} \leftarrow & \mathtt{any(s(J)),\ \ in\_G(Y,J),\ \ \neg t_2(Y,s(J)),\ \ \neg t_3(\_,Y,s(J))}. \\
\\
\mathtt{in\_FF(X,J)} \leftarrow & \mathtt{t_1(X,J)}. \\
\mathtt{in\_FF(X,s(J))} \leftarrow & \mathtt{any(s(J)),\ \ in\_F(X,J),\ \ \neg t_3(X,\_,s(J))}. \\
\\
\mathtt{sat_E(X,Y,J)} \leftarrow & \mathtt{t_3(X,Y,J)}. \\
\\
\mathtt{t_1(X,s(J))} \leftarrow & \mathtt{in\_H(J),\ \ in\_F(X,J),\ \ oldest\_in\_F(X,J)}. \\
\mathtt{t_2(Y,s(J))} \leftarrow & \mathtt{in\_G(Y,J),\ \ in\_H(J)}. \\
\mathtt{t_3(X,Y,s(J))} \leftarrow & \mathtt{in\_FF(X,J),\ \ in\_G(Y,J)}.
\end{array}
$$

□

The rule for a transition expresses the fact that if all its inputs become active at stage J, then the transition is enabled at stage s(J). For instance, transition t1 fires at stage s(J), if both places H and F are marked at stage J. The token obtained from place F is the one with the *oldest timestamp*. Thus, the *chronicle* parameter context is enforced.

An output token is created using the tokens received from the inputs of gate t3 and it is deposited into the place for E—see the rules for t3 and $\mathtt{sat_E}$.

This *component by component* translation can produce a lengthy set of rules. Moreover, observe that we have lost the direct correspondence between basic event occurrences and stages, which we have assumed so far.

We can solve these problems by providing a much simpler set of $Datalog_{1S}$ that simulates the sequence of *stable states.* A stable state is one that the Petri Net finally enters after an external event occurrence, and where no more transitions can fire. Thus, we obtain a one-to-one correspondence between basic event occurrences and stable states.

The sequence of stable states for the example of Table 4 is shown in Table 5. In the new table, every basic event occurrence moves the Petri Net to a new stable state, 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.

The following $Datalog_{1S}$ rules define the sequence of stages in Table 5.

Table 5. The simplified stage sequence for the execution sequence of Table 4.

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

EXAMPLE 17: The $Datalog_{1S}$ program modeling the stage sequence in Table 5.

$$
\begin{array}{ll}
\mathtt{in\_F(X,s(J))} \leftarrow & \mathtt{F(X,s(J)),\ in\_FF(\_,J).} \\
\mathtt{in\_F(X,s(J))} \leftarrow & \mathtt{any(s(J)),\ in\_F(X,J),\ \neg in\_FF(X,s(J)).} \\
\\
\mathtt{oldest\_in\_F(X,J)} \leftarrow & \mathtt{in\_F(X,J),\ \neg not\_oldest\_in\_F(X,J).} \\
\mathtt{not\_oldest\_in\_F(X,J)} \leftarrow & \mathtt{in\_F(Y,J),\ Y.Timestamp < X.Timestamp.} \\
\\
\mathtt{in\_FF(X,s(J))} \leftarrow & \mathtt{F(X,s(J)),\ \neg in\_FF(\_,J).} \\
\mathtt{in\_FF(X,s(J))} \leftarrow & \mathtt{sat_E(\_,\_,s(J)),\ in\_F(X,J),\ oldest\_in\_F(X,J).} \\
\\
\mathtt{sat_E(X,Y,s(J))} \leftarrow & \mathtt{G(Y,s(J)),\ in\_FF(X,J).}
\end{array}
$$

□

Regarding the final state of the Petri Net after processing a sequence of basic events, this program is *effect-equivalent* to the previous one. Its advantage is that it provides a compact semantics definition, without the need to simulate the exact execution semantics of the Petri Net, as the previous program did. For this reason, this approach is preferable for defining formally the semantics of various SAMOS constructs.

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.

Consider the two rules for `in_F` in the last program. The first rule says that if an event `F(x)` occurs at a time where other tokens already exist in place `F`, the token for `F(x)` will be placed in place `F` as well. The second rule (frame axiom) expresses the fact that after a new basic event occurrence, a token already in place `F` will remain there if it does not move to place `FF`, which is the only place that it can possibly move to.

The last program is *XY–stratified* [35]. Such programs can be evaluated in an incremental and efficient manner. In this case, 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 iterated fixpoint computation. After each basic event occurrence, the above computation steps are repeated, and a new stable state is obtained.

7. Conclusions

This paper has introduced the Event Pattern Language (EPL) for the specification of active database rules with composite temporal events. Furthermore, the paper has proposed a new approach for the formal definition of the semantics of such languages.

Whereas different operational formalisms have been used in the past for different languages, we have shown that the proposed logic-based method provides a general declarative framework, that unifies the semantic definitions of active database rules with composite temporal events. As a result, the task of specifying such languages and comparing their constructs and semantics is greatly simplified.

The unifying generality of this approach has been demonstrated by (i) providing a complete definition of the EPL language, (ii) clarifying some salient semantic differences among other languages such as ODE, SNOOP and SAMOS, and (iii) exposing some open issues in the definition of these languages in regard to parameterized events, negation and simultaneous events.

While the focus of the paper is on formal semantics issues, we have also described how $Datalog_{1S}$ rules can provide an effective and efficient basis for implementation. In particular, we outlined two approaches used at UCLA for the implementation of EPL prototypes. In the first approach, $Datalog_{1S}$ rules are viewed and implemented as FSM transitions. The second approach is based on the direct implementation of the derived rules in general-purpose deductive systems. By using the implementation mechanisms of deductive databases, problem-specific optimizations can be added. The main optimization consists in exploiting the particular form of these rules, whereby they can be implemented incrementally in a *history-less* manner [7]. This leads to implementation techniques similar to those used for supporting the class of XY-stratified programs in $\mathcal{LDL}$++. Forward-chaining production systems of more procedural nature, such as CLIPS can also provide a sound basis for implementation, due to their efficient rule activation mechanisms.

The approach proposed in this paper lays the seeds for further research. We have already studied the issue of temporal aggregation in active rules and the results can be found in [25].

An important issue yet to be answered is the formal comparison of different composite event specification languages. 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 [3]. Within this framework, the expressive power of such languages can now be studied using methods and tools previously used for Datalog and First-Order languages [1].

Another topic for further research that we are currently pursuing is that of alternative optimization strategies for composite event languages. For instance, while we have implicitly assumed that $Datalog_{1S}$ rules are evaluated incrementally as new basic events occur, composite events can also be evaluated and optimized as queries against a historical database of events.

Acknowledgments

The authors would like to thank Reza Sadri and the referees for their suggestions.

Notes

1. The syntax used for conditions is in fact that of the *where* clause in SQL; however, in EPL, the comma can also be used to denote the logical AND.
2. Note i that the *Timestamp parameter* is supplied by the database system, and thus, it is treated as a data parameter.
3. From now on, unless otherwise indicated, variables will denote tuples.
4. $(E_1, E_2, \ldots, E_{n-1}, E_n) = seq(E_1, seq(E_2, \ldots, seq(E_{n-1}, E_n)\ldots))$. Similarly for the relaxed sequence, the conjunction and the disjunction constructs.
5. 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 $*\mathtt{E} \equiv *\mathtt{seq}(\mathtt{E}, \varepsilon)$, where $\varepsilon \equiv$ no event.

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.
2. N. Arni, K. Ong, S. Tsur, and C. Zaniolo. LDL++: A second generation deductive database system. submitted for publication.
3. M. Baudinet, J. Chomicki, and P. Wolper. Temporal Deductive Databases. In Tansel et al. [32], chapter 13, pages 294–320.

4. 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.
5. 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, Sep. 1994.
6. S. Chakravarthy and D. Mishra. Snoop: An expressive event specification language for active databases. *Data and Knowledge Engineering*, 14(1):1–26, Nov. 1994.
7. J. Chomicki. History-less checking of dynamic integrity constraints. In *Proc. of the Intl. Conf. on Data Engineering*, pages 557–564, 1992.
8. R. David. *Petri Nets and Grafcet: Tools for modeling discrete event systems.* Prentice Hall, New York, 1992.
9. U. Dayal, A. Buchmann, and S. Chakravarthy. The HiPAC Project. In Widom and Ceri [34], chapter 7, pages 177–206.
10. U. Dayal, E.N. Hanson, and J. Widom. Active Database Systems. In W. Kim, editor, *Modern Database Systems.* Addison Wesley, 1995.
11. P. Deransart, M. Jourdan, and B. Lorho. *Attribute Grammars : definitions, systems, and bibliography.* Springer-Verlag, 1988.
12. S. Gatziu. *Events in an Active Object-Oriented Database System.* PhD thesis, Universität Zürich, 1994.
13. 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, Sep. 1993.
14. 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, February 1994.
15. 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.
16. 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.
17. G. Giuffrida and C. Zaniolo. EPL: Event Pattern Language. In *Proceedings of the third CLIPS Conference.* NASA's Johnson Space Center, September 1994.
18. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1988.
19. 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.
20. D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2:127–145, 1968.
21. J. W. Lloyd. *Foundations of Logic Programming.* Springer Verlag, 1977.
22. I. Motakis. *Temporal Reasoning in Active Databases.* PhD thesis, University of California, Los Angeles, 1997.
23. I. Motakis and C. Zaniolo. Composite Temporal Events in Active Database Rules: A Logic-Oriented Approach. In *Proceedings of the 4th Intl. Conference on Deductive and Object-Oriented Databases*, pages 19–37, 1995.
24. I. Motakis and C. Zaniolo. Composite Temporal Events in Active Databases: A Formal Semantics. In J. Clifford and A. Tuzhilin, editors, *Recent Advances in Temporal Databases*, Workshops in Computing Series, pages 332–351. Springer, Sep. 1995.

25. I. Motakis and C. Zaniolo. Temporal Aggregation in Active Database Rules. In *Proceedings of the ACM SIGMOD Intl. Conf. on Management of Data*, May 1997.
26. NASA, Lyndon Johnson Space Center, Software Technology Branch. *CLIPS 6.0 Reference Manual*, June 1993.
27. N. W. Paton, J. Campin, A. A. Fernandes, and M. H. Williams. Semantics for Active Database Systems. In *Proceedings of the 2nd International Workshop on Rules in Database Systems*, pages 21–35, 1995.
28. P. Picouet and V. Vianu. Semantics and Expressiveness Issues in Active Databases. In *Proceedings of the 14th Symposium on Principles of Database Systems*, pages 126–138, 1995.
29. 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):35–51, March 1994.
30. W. Reisig. *Petri Nets: An Introduction.* Springer-Verlag, Berlin, 1985.
31. A. P. Sistla and O. Wolfson. Temporal Conditions and Integrity Constraints in Active Database Systems. In *Proc. of the 1995 ACM SIGMOD Intl. Conference on Management of Data*, pages 269–280, 1995.
32. A. Tansel et al., editors. *Temporal Databases: Theory, Design and Implementation.* Benjamin/Cummings, 1993.
33. J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Volumes I and II.* Computer Science Press, 1989.
34. J. Widom and S. Ceri, editors. *Active Database Systems: Triggers and Rules for Advanced Database Processing.* Morgan Kaufmann, 1996.
35. 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.
36. C. Zaniolo, S.Ceri, C. Faloutsos, R. Snodgrass, V.S. Subrahmanian, and R. Zicari. *Advanced Database Systems.* Morgan Kaufmann Publishers, 1997.

Contributing Authors

Iakovos Motakis obtained his Diploma in Electrical Engineering from the National Technical University of Athens, in 1989, and his M.S. and Ph.D. from U.C.L.A., in 1992 and 1997, respectively.

He is currently a consultant with Cambridge Technology Partners, where he specializes in data warehouse and data mining applications. He was previously with NCR Teradata. His current research interests also include active and temporal reasoning in databases.

Carlo Zaniolo is a professor of Computer Science with the UCLA School of Engineering and Applied Sciences, where he occupies the N.E. Friedmann Chair in Knowledge Science.

Before joining UCLA, in 1991, C. Zaniolo earned twenty years of industrial experience. He was an associate director in the Software Technology Program of MCC (Austin, Texas); before MCC he was with AT&T Bell Labs (Murray Hill, NJ), Sperry Research (Sudbury, Massachusetts) and Burroughs Corporation (Pasadena, California).

Dr. Zaniolo's research interests include database systems, non-monotonic reasoning in databases and knowledge bases, and document servers for the information age. He served as program chairman of several conferences, including ACM SIGMOD and VLDB conferences.